Düzce University Journal of Science & Technology

Research Article

Discovering Sequential Source Code Patterns in Software Engineering

Kökten Ulaş BİRANT [a,*], Dilara KIRNAPCI [b]

[a] *Department of Computer Engineering, Dokuz Eylul University, Izmir, TURKEY*
[b] *The Graduate School of Natural and Applied Sciences, Dokuz Eylul University, Izmir, TURKEY*
** Corresponding author's e-mail address: ulas.birant@ deu.edu.tr*
DOI: 10.29130/dubited.905510

ABSTRACT

Discovering sequential patterns in source codes is an important issue in software engineering since it can provide useful knowledge to help in a variety of tasks such as code completion, code refactoring, developer profiling, and code complexity measurement. This paper proposes a new framework, called *Source Code Miner* (SCodeMiner), which discovers frequent sequential rules within a software project. The proposed framework firstly transforms a Java code into a sequence data and then applies a sequential pattern mining (SPM) algorithm. This study is also original in that it compares four SPM algorithms in terms of computational time, including sequential pattern discovery using equivalence classes (SPADE), prefix-projected sequential pattern mining (PrefixSpan), bi-directional extension (BIDE+), and last position induction (LAPIN). The experiments that carried out on an open-source software project showed that the proposed SCodeMiner framework is an effective mining tool in identifying coding patterns.

Keywords: *Source code patterns, Sequential pattern mining, Software engineering, Source code analysis*

Yazılım Mühendisliğinde Sıralı Kaynak Kodu Modellerini Keşfetme

ÖZ

Kaynak kodlardaki sıralı örüntüleri keşfetmek yazılım mühendisliğinde önemli bir konudur, çünkü kod tamamlama, kodu yeniden düzenleme, geliştirici profili oluşturma, ve kod karmaşıklığı ölçümü gibi çeşitli işlemlerde yardımcı olacak yararlı bilgiler sağlayabilmektedir. Bu makale, bir yazılım projesinde sıkça geçen sıralı kuralları keşfeden ve *Kaynak Kod Madencisi* (SCodeMiner) adı verilen yeni bir yazılım çerçevesi önermektedir. Önerilen yazılım çerçevesi ilk olarak bir Java kaynak kodunu bir sıralı veri tabanına dönüştürür ve ardından bir sıralı örüntü madenciliği (SPM) algoritması uygular. Bu çalışma aynı zamanda, dört SPM algoritmasını çalışma süresi açısından karşılaştırması açısından da orijinaldir. Bu algoritmalar şunlardır: ön ek ile öngörülen sıralı örüntü madenciliği (PrefixSpan), denklik sınıflarını kullanarak sıralı örüntü keşfi (SPADE), çift yönlü uzatma (BIDE+), ve son pozisyon indüksiyonu (LAPIN). Açık kaynak kodlu bir yazılım projesi üzerinde gerçekleştirilen deneyler, önerilen SCodeMiner yazılım çerçevesinin kodlama örüntülerini belirlemede etkili bir madencilik aracı olduğunu göstermektedir.

Anahtar Kelimeler: *Kaynak kod örüntüleri, Sıralı örüntü madenciliği, Yazılım mühendisliği, Kaynak kod analizi*

Received: 29/03/2021, Revised: 16/08/2021, Accepted: 22/08/2021

I. INTRODUCTION

One of the main concerns in the software engineering field is the improvement of productivity and quality during software development. In this context, code analysis plays an important role since it can be useful in finding previously unknown, potentially valuable, hidden, and accurate coding patterns. *Source Code Analysis* (SCA) is the process of automatically extracting required information about software from its source code in the software engineering process [1]. SOC is an essential step of a code review that focuses on evaluating, monitoring, and improving software quality. *Source Code Review* (SCR) is a software development activity, where a code is automatically examined for a particular purpose such as for checking coding standard or structured design, controlling logical correctness, or discovering faults [2]. It is a non-trivial and effective way to decrease the price and time of the software product and improve overall productivity.

A *Coding Pattern* is a typical sequence of programming statements that has a particular behavior. Since coding patterns indicate hidden rules in a software project, discovering them can help software engineers in various ways such as (i) improving developer performance via code completion [3, 4], (ii) comparing code versions to perform appropriate modifications and enhancements [5], (iii) profiling developers, (iv) detecting crosscutting concerns, and (iv) measuring code complexity. In this work, we used the sequential pattern mining technique to discover coding patterns in an efficient way.

Pattern Mining is a fundamental task in data mining, with the goal of discovering frequently recurring patterns in a database [6]. A *pattern*, which is in the form of *<pattern>:support*, consists of the content of the pattern and the number of the transactions which have the pattern. *Sequential Patterns* are formed from ordered sequences of elements that frequently occur in a dataset. *Sequential Pattern Mining* (SPM) is one of the well-known data mining methods utilized to identify any specific order of occurrences [7]. A *sequence* is a typical ordered list of elements (i.e., events). A typical example of a sequence of events is a sequence of items bought by a particular customer at different times in order over an interval. For example, if there are two sequences *<a b c>* and *<a d b>* that have three elements, the pattern *<a b>* is detected since it appears two times. SPM discovers the complete collection of frequent sub-sequences from a given sequence dataset. SPM is a significant task that has been extensively studied in many applications such as medical record analysis, web-log analysis, and market basket analysis. Several types of patterns may be detected from the datasets, i.e., frequent sequences, periodic patterns, and sequential rules.

When we provide source code files to a sequential pattern mining algorithm as input, source codes are organized in sequential transactions and then the coding patterns can be extracted to discover useful knowledge about the software. Here, a key challenge is to automatically transform a source code into sequences of programming statements using a tokenization approach. Another challenge is the selection of an SPM algorithm by considering both the characteristics of the dataset and the key features of the algorithms together. The performance of the algorithm highly depends on the number of pattern candidates, the length of the sequences, and the number of distinct items in a dataset. An efficient SPM algorithm should be used to speed up the computations.

The main contributions of this study can be listed as follows. (i) This paper proposes a novel framework, *Source Code Miner* (SCodeMiner), which discovers frequent sequential rules within software projects. (ii) This study is also original in that it compares four SPM algorithms in terms of computational time, including prefix-projected sequential pattern mining (PrefixSpan) [8], sequential pattern discovery using equivalence classes (SPADE) [9], bi-directional extension (BIDE+) [10], and last position induction (LAPIN) [11].

The experiments that carried out on an open-source software project showed that the proposed SCPM framework is an effective mining tool in identifying coding patterns. Therefore, it can be successfully used for software engineering projects. The proposed SCodeMiner framework can be useful for

developers, programmers, and testers as well as other professionals involved in other aspects of software engineering.

This article is organized into five sections as follows. Section 2 discusses studies related to our work. The proposed framework is detailed in Section 3. The experiments are given in Section 4, before concluding the main findings in Section 5.

II. RELATED WORK

Analysis of source codes to discover patterns allows software engineers to identify frequently occurring sequences within software projects. For this purpose, some data mining studies have been conducted recently. Some previous studies [3]-[5], [12]-[23] on source code analysis are given in Table 1. A variety of data mining algorithms have been used for source code analysis such as graph neural network (GNN) [12], multi-layer perceptron (MLP) [5, 15], restricted Boltzmann machine (RBM) [4], naive Bayesian (NB) [5, 15], Bayesian networks (BN) [3], logistic regression (LR) [5, 15], and support vector machine (SVM) [15]. Some deep learning techniques have also been successfully applied to source code analysis, including long short-term memory (LSTM) [4, 13, 15, 17], bidirectional long short-term memory (BLSTM) [13, 15, 17], gated recurrent units (GRU) [13, 17], bidirectional gated recurrent unit (BGRU) [13], deep belief network (DBN) [4], recurrent neural networks (RNN) [4, 12, 17], and convolutional neural networks (CNN) [4, 12, 15, 17]. The sequential pattern mining algorithms that have been used in source code analysis are Generalized Sequential Pattern (GSP) [20], PrefixSpan [16, 18, 19], BIDE [22], and Pre-order Linked Web Access Pattern-Tree (PLWAP) [21]. Some ensemble learning algorithms have also been used in source code analysis such as Random Forest (RF) [4, 5, 15], AdaBoost (AB) [5], and Gradient Boosting Decision Tree (GBDT) [15].

In the literature, previous studies on source code analysis have been focused on either classification task [4, 5, 12, 13, 15, 17] or pattern mining task [3, 14, 16], [18]-[23]. Until now, data mining techniques have been used to analyze source codes written in a variety of programming languages such as C [4, 13, 14, 15], C++ [12, 14], Java [5, 14], [18]-[21], PHP [17], and Python [16]. In this study, we focus on the Java programming language. The codes of various open-source software projects have been analyzed, such as Eclipse, Linux Kernel, JFreeChart, Dnsjava, and JmDNS. In addition, some publicly available datasets collected to evaluate the vulnerability of source codes have been used in the previous works such as the software assurance reference dataset (SARD) [13, 15, 17] and national vulnerability database (NVD) [13, 15].

Newman et al. [14] investigated frequent naming patterns in different identifier types, such as attribute and class names. Date et al. [19] reported the characteristics of coding patterns over versions. They extracted coding patterns of each program version, and then investigated the number of versions in which the coding patterns appear.

Akbar et al. [20] applied a method categorization technique to mine API usage patterns for the purpose of improving code completion. Kagdi et al. [24] proposed an approach to detect the call-usage patterns and variable locations by source code analysis using sequential pattern mining algorithms. They applied the SPADE algorithm to the Apache HTTPd v2.0.55 system source code. In another study [18], a software clone detection method was proposed utilizing a maximal frequent SPM method. The source codes of the Apache Struts 2.5.2 Core project were used for extracting code-matching statements in the Java programming language. A plug-in, called Vertical Code Completion was implemented for the Eclipse IDE [21]. The suggestions of new code sequences were made based on the patterns obtained by using the PLWAP Algorithm, which is one of the SPM algorithms. Takei and Yamana [22] extended the bi-directional execution (BIDE) algorithm by adding an intensity constraint (IC) and then used the proposed IC-BIDE algorithm for coding pattern extraction. They applied the algorithm on Bullet Physics, MySQL, and OpenCV source codes.

Table 1. *Comparison of source code analysis studies (C: Classification P: Pattern mining).*

Ref	Year	Description	Task		Methods	Language	Project
			C	P			
[12]	2021	Vulnerability detection	√		GNN, CNN, RNN	C/C++	Linux Kernel FFmpeg Wireshark Libav
[13]	2021	Vulnerability detection	√		LSTM, BLSTM, GRU, BGRU	C	SARD NVD
[4]	2020	Code completion	√		LSTM, RNN, CNN, RF, RBM, DBN	C	An online judge (AOJ) System
[14]	2020	Patterns related to source code identifiers		√	Part-of-speech (POS) tagging	C, C++, Java	20 open-source systems
[15]	2020	Vulnerability detection	√		LR, NB, SVM, MLP, GBDT, RF, CNN, LSTM, BLSTM	C	SARD NVD
[5]	2019	Analysis of code versions	√		NB, MLP, AB, RF, LR	Java	JFreeChart Heritrix
[16]	2019	Code review		√	PrefixSpan	Python	OpenStack
[17]	2019	Vulnerability detection	√		LSTM, BLSTM, CNN, GRU, RNN	PHP	SARD SQLI-LABS
[18]	2016	Finding Software Clones		√	Apriori, PrefixSpan	Java	Apache Struts 2.5.2 Core
[3]	2015	Code completion		√	BN	OOP	Eclipse
[19]	2015	Analysis of code versions		√	PrefixSpan	Java	10 open-source programs
[20]	2014	Code completion		√	GSP	Java	10 open-source projects
[21]	2014	Code completion		√	PLWAP	Java	Ant Eclipse Maven Log4j
[22]	2013	Coding Pattern Extraction		√	BIDE	Various	Bullet Physics MySQL OpenCV
[23]	2012	Analysis of code versions		√	PrefixSpan	Java	Dnsjava JmDNS
Proposed Approach		Code analysis for general-purpose		√	PrefixSpan, SPADE, BIDE+, LAPIN	Java	Apache Tomcat

Code review is an important step in software development and includes source code verification, modification, and feedback. Ueda et al. [16] detected similar code changes patterns that commonly appear in the project history. They applied a sequential pattern mining algorithm to the OpenStack project and detected 1476 improvement patterns from the Python source codes. Kim et al. [25] discovered coding patterns with their characteristics by performing an evaluation. They selected several indicators and performed an analysis by investigating relations between the characteristics and values of the patterns. Ishio et al. [26] focused on mining the code patterns related to method calls.

While some of the previous studies [14, 20, 26] discovered the coding-related patterns from a single version of a software project, some studies [19, 23] investigated the patterns in multiple versions of the project. In the former one, patterns can be detected only from a particular version of the source code.

However, in the latter one, coding patterns were extracted from a single version separately, and after that, the common patterns were searched in multiple versions.

Our study differs from the aforementioned studies in that it proposes a new framework that especially focuses on nested loop and control statement blocks. Furthermore, it uses an efficient sequential pattern-mining algorithm to provide increased computational performance. Moreover, our method can be used for general-purpose since we are concerned with examining the general structure of the source. On the other hand, the existing approaches typically proposed for a specific purpose such as for source code identifiers [14], software versions [5, 19, 23], API usage patterns [20], call-usage patterns [24], software clones [18], code review [16], crosscutting concerns [26], and vulnerability detection [12, 13, 15, 17].

III. MATERIAL AND METHODS

A. PROPOSED FRAMEWORK

In this study, we propose a novel framework: *Source Code Miner* (SCodeMiner), which discovers frequent sequential rules within software projects. Figure 1 shows the general architecture of the proposed framework. Before the data mining techniques, a source code available in a repository has to be processed to transform it into a proper form. For this reason, the framework basically consists of two main phases: data preprocessing and pattern mining. In the *data preprocessing* phase, the source code is converted into sequences of programming statements. In the first step of this phase, a source code is taken from the repository to be analyzed. In the next step, a parser is used for tokenization which extracts the statements with various levels of nesting and identifies tokens, especially including loop and control statements. Here, all keywords, delimiters, and procedures/functions in the source code are obtained separately while comment lines are ignored. After that, a sequence generator converts tokens into sequences and stores them in sequence data. In the *pattern mining* phase, a SPM algorithm is utilized to extract frequent subsequences from a collection of sequences. Here, each pattern is a sequence of code elements. Each pattern is evaluated by a minimum support threshold where the support value of a sequence s_i in the dataset D is the number of sequences $s \in D$ that contains s_i. The discovered patterns are stored in a database and presented to the user in an appropriate form through an application.

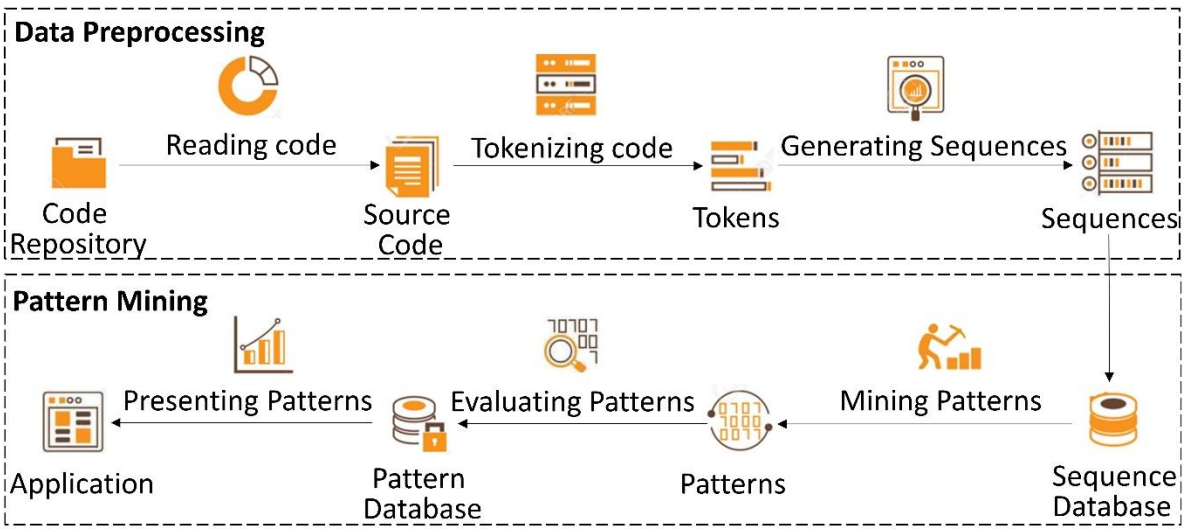

Figure 1. *The general architecture of the proposed SCodeMiner Framework.*

B. DATA PREPROCESSING

Each source code file may include more than one class, each class may involve a set of methods and each method may consist of various kinds of blocks. A block in the source code is represented by the "{" and "}" symbols. The framework firstly extracts all the blocks and then captures the sequences of programming tokens. Inside each code block, more than one programming statement can be existing and all of them are listed in a sequence in the order they present in the block. In other words, all programming statements that exist in the same block are collected in the same sequence. In order to increase the computational power, tokens (items) in sequences are assigned with a unique index by utilizing a mapping method.

Figure 2 shows three example sequences extracted from different source code fragments. The following keywords are specially processed by a parser: *if, else, for, while, get, set, break, math,* and *return*. The sequences are created by analyzing tokens using a set of rules and stack data structure. Some keywords such as "*return*" and "*break*" are directly included in the sequence. An "*if-else*" statement is converted into a series of "*IF*", "*ELSE*", and "*ENDIF*" terms. The code elements controlling by the "*if*" statement are put between the "IF" term and its corresponding "*ENDIF*" term. The "*for*" and "*while*" statements are translated into a pair of "*LOOP*" and "*ENDLOOP*" terms. In order to represent nested *if* and *loop* statements, the "*ENDIF*" and "*ENDLOOP*" keywords are added to the sequences. The basic mathematical operations such as addition or subtraction and the subroutines in the math library are translated as the "*MATH*" keyword. The user-defined function calls are stated as the "*FUNC*" keyword. The get and set function calls are separated from other user-defined function calls and the "*GET*" and "*SET*" specific keywords are used to indicate them. The framework ignores all primitive data types (i.e., *int, string, char*), variables, and comment statements.

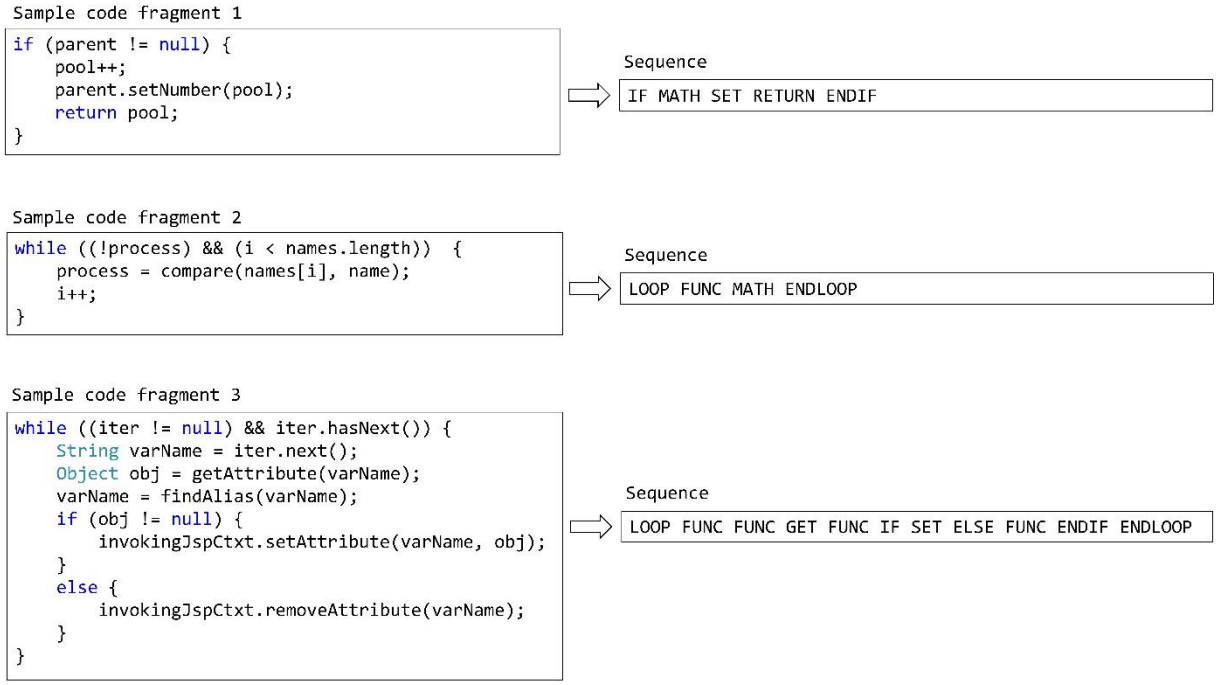

Figure 2. *Sample sequence generation from source code fragments.*

C. PATTERN MINING

Assume that $I = \{ i_1, i_2, ..., i_n \}$ be a finite set of elements called items. An itemset is a nonempty set of items. A sequence is a finite ordered and consecutive list of item-sets. A typical sequence *s* is stated as

$<t_1\ t_2 \ldots t_m>$, where t_j is an itemset, $t_j \subseteq I$. In this context, t_j can be also named an element of the sequence which is stated as $(x_1, x_2, \ldots, x_r)$, where x_k is an item, $x_k \in I$. In our study, the brackets are ignored since each element has only a single item, i.e., element (x) is illustrated as x. Note that an item may exist multiple times in different parts of the same sequence; however, it can appear at most once in an element of a sequence. The length of *a* sequence, denoted by *l (s)*, is the number of items in the sequence such that l (s) = $|t_1| + |t_2| + \ldots + |t_m|$, i.e., $l(s) = \sum_{i=1}^{m} |t_i|$. For instance, the length of the sequence <(1 2 3)(1 4)(2 5)> is 7. A sequence with length *l* is called an *l-sequence*. The size of a sequence, denoted by $|s|$, is the number of itemsets in the sequence. A sequence $\alpha = <a_1\ a_2 \ldots a_u>$ is called a sub-sequence of another sequence $\beta = <b_1\ b_2 \ldots b_v>$ if there exist numbers $1 \leq j_1 < j_2 < \ldots < j_u \leq v$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, ..., $a_u \subseteq b_{ju}$. When a sequence α is included in another sequence β, then α is called as a subsequence of β, and β is called to be a supersequence of α. For instance, the sequence β = <(1 2 3)(3 4)> is a super-sequence of α = <(1 2)(4)>.

A sequence database, denoted by $D = \{ s_1, s_2, \ldots, s_n \}$, is a finite set of sequences, and $|D|$ refers to the number of sequences in *D*. Given a database *D* and user-determined minimum support (*minsup*), the algorithm extracts all the sub-sequences with *support* ≥ *minsup*. The *support* of a sequence *s* in *D*, denoted by *sup(s)*, is the number of transactions in *D* to which *s* belongs. For example, if we have two sequences <*abc*> and <*ace*>, the support of the pattern "*ac*" is 2 since it appears two times. Given a positive number *minsup*, a sequence *s* is called as frequent pattern or frequent sequence if *sup(s)* ≥ *minsup*. The projected dataset of a sequence α contains all the suffixes of sequences that include α.

Table 2 gives a sample sequence dataset in the first row and then shows the discovery of patterns step by step. Each row of the table illustrates a *k*-length pattern (also called prefix), its projected dataset, and new (*k + 1*)-length patterns. The symbol *Φ* represents an empty projected dataset. As shown in Table 2, the PrefixSpan algorithm repeatedly generates prefix-projected itemsets to discover frequent patterns. A typical pattern and its corresponding support value are represented in the form <*pattern*> : *support*. The algorithm firstly finds 1-length patterns which are the following in this example: <*IF*>: 3, <*FUNC*>: 2, <*MATH*>: 4, and <*ENDIF*>: 3. Here, the pattern <*MATH*>: 4 was extracted from the dataset, instead of <*MATH*>: 5 since the traditional SPM methods ignore the number of occurrences of an item within one transaction. The occurrences of items in one transaction are limited to binary values (i.e., each item may or may not appear in one transaction). The support value of an item is the number of transactions that contain this item. In other words, the support of item i_x in a database *D* is the number of transactions in *D* to which belongs the item i_x. The number of transactions is 4 in this example and *MATH* appears at least one time in all transactions. It may be noted here that high-utility sequential pattern (HUSP) mining has emerged as a new research topic, which considers the items that may appear zero, once, or multiple times in one transaction in a quantitative sequence database [27]. Since the minimum support is 50% (*minsup* = 2) in this example, the algorithm filters out "*ELSE*", "*LOOP*", "*ENDLOOP*", "*SET*", and "*RETURN*" elements because each one is involved in only one sequence.

The algorithm repeatedly finds (*k+1*)-*length* patterns by considering *k-length* patterns. The method finds all the sequences containing an itemset and generates a projected dataset in which each sequence is prefixed with the first occurrence of the itemset. For instance, <*FUNC*>-projected dataset includes two sequences: <*MATH ENDIF*> and <*MATH ENDLOOP*>. Frequent itemsets in a projected dataset represent (*k + 1*)-*length* patterns. For example; the "*MATH*" element in <*FUNC*>-projected dataset results in its corresponding *2-length* patterns: <*FUNC MATH*> : 2. Each prefix-projected itemset is evaluated by a minimum support (*minsup*) threshold. If the support of an itemset is less than *minsup*, then this itemset is eliminated. For example, when considering the prefix <*MATH*>, the algorithm filters out <*MATH ELSE MATH ENDIF*>: 1, <*MATH ENDLOP*>: 1, and <*MATH SET RETURN ENDIF*>: 1 candidate itemsets since the minimum support is 50% (*minsup* = 2) in this example and each one is involved in only one transaction. On the other hand, the pattern <*MATH ENDIF*>: 3 is extracted since it appears in three transactions and so its support is greater than the *minsup*. The algorithm finishes when no new pattern is found in an iteration. Eventually, the algorithm with *minsup* = 2 finds five frequent patterns from the sample dataset: <*IF MATH*>: 3, <*IF ENDIF*>: 3, <*FUNC MATH*>: 2, <*MATH ENDIF*>: 3, and <*IF MATH ENDIF*>: 3.

Table 2. *A sample sequence dataset and the discovery of patterns from it.*

Prefix	Prefix-Projected Itemsets	Patterns
Original dataset	<IF FUNC MATH ENDIF> <IF MATH ELSE MATH ENDIF> <LOOP FUNC MATH ENDLOOP> <IF MATH SET RETURN ENDIF>	<IF>: 3 <FUNC>: 2 <MATH>: 4 <ENDIF>: 3
<IF>	<FUNC MATH ENDIF> <MATH ELSE MATH ENDIF> <MATH SET RETURN ENDIF>	<IF MATH>: 3 <IF ENDIF>: 3
<FUNC>	<MATH ENDIF> <MATH ENDLOOP>	<FUNC MATH>: 2
<MATH>	<ENDIF> <ELSE MATH ENDIF> <ENDLOOP> <SET RETURN ENDIF>	<MATH ENDIF>: 3
<ENDIF>	∅	
<IF MATH>	<ENDIF> <ELSE MATH ENDIF> <SET RETURN ENDIF>	<IF MATH ENDIF>: 3
<IF ENDIF>	∅	
<FUNC MATH>	<ENDIF> <ENDLOOP>	
<MATH ENDIF>	∅	
<IF MATH ENDIF>	∅	

Patterns extracted by the *SCodeMiner* framework have the following properties:

• A coding pattern is a sequence of programming elements and statements. In other words, a coding pattern is a list of tokens and each token corresponds to a coding element or statement.
• Each item in a pattern has at least *minsup* value. Here, the term "*item*" represents a special programming element in a code fragment corresponding to the pattern.
• A pattern can comprise a different number of code elements. The *length* of a pattern is the number of items in the pattern. For example, the <*FUNC MATH*> pattern is a 2-length pattern, while the <*IF MATH ENDIF*> pattern is a 3-length pattern.
• When a pattern appears in the code fragments with minimum support or higher, it is said to be frequent.
• A pattern implies its sub-patterns that have a fewer number of items. For instance, a pattern <*GET FUNC MATH RETURN*> implies four sub-patterns comprising 3-items: <*GET FUNC MATH*>, <*GET FUNC RETURN*>, <*GET MATH RETURN*>, and <*FUNC MATH RETURN*>.

D. ALGORITHMS

Over the last decade, different algorithms have been proposed in the field of SPM, each of which has different properties [28, 29]. In this study, we chose the PrefixSpan [8], SPADE [9], BIDE+ [10], and LAPIN [11] algorithms based on important key features supported by these methods (Table 3). Since each algorithm uses a different approach (apriori-based, pattern-growth, constraint-based, and early-pruning), we can perform a comparative analysis of their performances on the dataset. *Apriori-based approaches* scan the original dataset several times to extract frequent itemsets of size *k* at each *k*th-iteration, while the *pattern-growth techniques* build a representation of the dataset and then provide a way to partition the search space. *Early-pruning approaches* rely on position induction to avoid support counting and to prune candidates at the very early stage of the mining process as much as possible.

Constraint-based approaches focus on identifying the entire set of patterns satisfying a particular constraint *C* to reduce the number of retrieved patterns by pruning uninteresting ones.

Table 3. *The key features of the sequential pattern mining algorithms.*

Algorithms / Properties	PrefixSpan	SPADE	BIDE+	LAPIN
Approach	Pattern-Growth	Apriori-Based	Constraint-Based	Early-Pruning
Database Layout	Horizontal	Vertical	Horizontal	Vertical
Traverse	Depth-First Search	Breadth-First Search	Depth-First Search	Depth-First Search
Search	Top-Down	Bottom-Up	Bi-directional	–
Monotone	Prefix-monotone	Anti-monotone	Prefix-monotone	Anti-monotone
Generate-and-Test	X	√	X	X
Candidate Pruning	√	√	√	√
Search- Space Partitioning	√	√	√	√
Single Scan of Database	√	X	X	√
Prefix Growth	X	X	X	√
Position Induction	X	X	X	√
Memory-only	√	X	√	√
Compression or/and Sampling	X	X	X	X
Constraints and Taxonomies	X	X	√	X
Counting Support Without Scanning	X	X	X	√
Pros	- Scanning the original dataset once - Effective when low support thresholds are used	Thanks to lattice-theoretic approach, good for fast mining in large datasets	- Executing some checking steps to avoid maintaining -Memory efficiency	- Reducing the search space -Effective in mining dense datasets
Cons	Creation and analyzing of a large number of projected sub-datasets	Inefficient for mining long sequential patterns	Multiple scans (closure checking, back-scan, and scan-skip)	Additional computation time and storage space to convert a dataset from horizontal to vertical format

Some SPM algorithms like SPADE and LAPIN use a vertical form of the dataset rather than the regular horizontal layout. The traversal in the search space (Depth-First Search (DFS) or Breadth-First Search (BFS)) makes a big difference in performance. The *Generate-and-Test* feature implies using exhaustive join operators such that the pattern is basically grown one element at a time and tested against the minimum support. *Prefix-monotone* property indicates that if for each sequence *S* that satisfies a constraint *C*, so does every sequence having *S* as a prefix, while *Anti-monotone* property states that every non-empty subsequence of a pattern is also a sequential pattern.

PrefixSpan is considered one of the efficient SPM algorithms since it scans the original dataset one time, usually along with search space partitioning and candidate pruning. In this study, we especially chose the PrefixSpan and SPADE algorithms since a comprehensive performance study [8] showed that they outperformed the other alternative algorithms such as FreeSpan and GSP in terms of both memory usage and running time. As reported in [30], PrefixSpan is faster than other hybrid or pure pattern-growth methods, like PLWAP, although it is less memory-efficient. We also applied the BIDE+ algorithm on the dataset due to its advantages such as no need to generate candidate sequential patterns. Besides, the LAPIN algorithm has the advantages of reducing the search space during the sequential mining process and being effective in mining dense databases [11].

Since the characteristics of the data (i.e., the length of the sequences, the number of distinct items, dense vs. sparse) and input parameter settings (i.e., minimum support threshold) have an important impact on the performance of the algorithm, we tested and compared four algorithms to determine the best one. Therefore, a combination of theoretical analysis and empirical evaluation was used to determine the best algorithm for the given dataset.

IV. EXPERIMENTAL STUDIES

In the experiments, we obtained the results with the *PrefixSpan* algorithm by utilizing the SPMF open-source library [31]. The library can be freely downloaded from the website http://www.philippe-fournier-viger.com/spmf/. In addition, we compared four different sequential pattern mining algorithms in terms of runtime, including PrefixSpan [8], SPADE [9], BIDE+ [10], and LAPIN [11]. In each experiment, the SPM algorithms were executed 5 times and the average values were reported here. All experiments presented in this study were conducted on a laptop equipped with Intel Quad-Core 2.7 GHz CPU and 8 GB of RAM.

A. DATASET DESCRIPTION

In the experimental studies, we used the Apache Tomcat[1] v9.0.28 open-source software project for detecting coding patterns. The Apache Tomcat is an open-source implementation of the JavaServer Pages, Java Expression Language, Java WebSocket technologies, and Java Servlet. It contains 2460 java files. In the data preprocessing phase, these source code files were read and translated into sequences. Total, 17220 sequences were obtained by converting 2460 java files.

B. EXPERIMENTAL RESULTS

We carried out four experiments for the following purposes: (i) to find frequent coding patterns, (ii) to investigate the relation between the *minsup* settings and the number of frequent patterns, (iii) to explore the distribution of k-length patterns, and (iv) to compare four alternative SPM algorithms.

In the first experiment, we discovered the frequent coding patterns by running the PrefixSpan algorithm with %1 minimum support threshold. Table 4 shows sample patterns. The results show that source codes usually contain frequent patterns of subroutine calls, mathematical operations, control flows, and loops. For example, the pattern <*IF GET ELSE GET ENDIF*> indicates that 4.18% of the code blocks have a get function inside an "*if-else*" control statement. The example pattern with ID 18 contains an "*IF ELSE IF*" statement block, while the last sample pattern includes a nested-IF block. Some programming elements such as "*THROW*" may not involve in the list of frequent patterns due to their sparsity in the source codes. When a low minimum support threshold is chosen, rare programming elements can also be extracted. However, in this case, a huge number of patterns is obtained, which increases the runtime of the algorithm and leads to producing some uninteresting patterns.

[1] http://github.com/apache/tomcat

Table 4. *Sample discovered sequential patterns along with their lengths, frequencies, and support values.*

ID	Pattern	Length	Frequency	Support *(%)*
1	IF	1	15133	87.88
2	FUNC	1	11916	69.20
3	LOOP	1	3485	20.24
4	MATH	1	3944	22.90
5	GET	1	6852	39.79
6	SET	1	1738	10.09
7	FUNC RETURN	2	2369	13.76
8	MATH RETURN	2	580	3.37
9	IF GET ENDIF	3	6187	35.93
10	IF SET ENDIF	3	1591	9.24
11	LOOP MATH ENDLOOP	3	2026	11.77
12	LOOP FUNC GET ENDLOOP	4	1316	7.64
13	IF MATH SET ENDIF	4	249	1.45
14	LOOP FUNC GET FUNC ENDLOOP	5	944	5.48
15	IF GET ELSE GET ENDIF	5	720	4.18
16	IF FUNC RETURN ELSE RETURN ENDIF	6	266	1.54
17	LOOP FUNC IF BREAK ENDIF ENDLOOP	6	201	1.17
18	IF FUNC ELSE IF FUNC ENDIF	6	570	3.31
19	LOOP IF FUNC ENDIF MATH FUNC ENDLOOP	7	234	1.36
20	IF FUNC LOOP IF FUNC ENDIF ENDLOOP ENDIF	8	242	1.41
21	IF IF FUNC MATH IF ENDIF FUNC ENDIF ENDIF	9	182	1.06

The coding patterns found can help software engineers in various ways. Code completion can be provided to the developers according to the patterns, invoking automatically or by pressing a key. When a programming element is initiated by the user, the possible completion proposals according to the frequent coding patterns can be shown through a popup menu. Furthermore, coding patterns discovered from different versions of software projects can be compared to uncover changes. It can be utilized for the interpretation of software versions, modeling changes, and understanding the modifications and enhancements. Moreover, the coding patterns can be used for developer profiling. For example, the developer's expertise or skill level can be predicted according to the patterns. Similarly, based on their coding patterns, the developers can be ranked in terms of ability. In addition, the coding patterns can give an idea about code quality and code complexity. As the source code gets more complex, sequences in the dataset contain longer patterns.

In the second experiment, the relation between the minimum support settings and the number of frequent patterns was investigated. The algorithm was run on the sequence dataset with support values varying from 10% to 50% with an increment of 5%. Figure 3 presents the number of discovered frequent patterns along with the minimum support threshold. Frequent patterns were obtained by the subsequences whose support value was greater than or equal to the threshold value. However, the other subsequences that had a lower support value than the threshold value were discarded. As can be seen in Figure 3, the number of frequent patterns decreases exponentially when the minimum support threshold increases. For example, when the minimum support value was 10%, the algorithm discovered 155 frequent coding patterns, whereas it found 15 patterns for the minimum support of 35%. Therefore, it is possible to say that we can obtain a huge number of frequent patterns by executing the algorithm with small support thresholds. However, using a low minimum support threshold may result in generating too many patterns including a lot of uninteresting rules. In addition, it leads to increasing computational complexity and memory requirements. On the other hand, a high *minsup* value can lead to miss useful patterns. For this

reason, an appropriate minimum support threshold should be set to efficiently discover valuable and interesting frequent patterns.

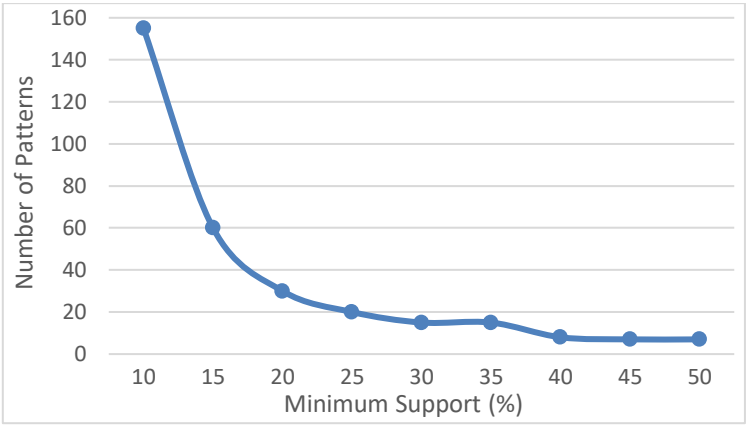

Figure 3. *Numbers of retrieved frequent sequences*

In the third experiment, the distribution of *k*-length patterns was investigated by running the algorithm with different minimum support values ranging from 5 to 15. Table 5 presents the numbers of discovered patterns separately, varying from 1-length to 8-length patterns. Hence, it shows the effect of support threshold value on the length of frequent patterns. The results show that the number of patterns obtained from the dataset was high when a low minimum support threshold was determined. For example, the number of 4-length patterns is 234 when *minsup* = 5%, while the number of 4-length patterns was 10 when *minsup* = 15%. In this way, the frequent coding patterns can be found in a more manageable size. It can be observed from Table 5 that the number of 3-length and 4-length patterns is usually higher than others. The results construct a form quite similar to the bell curve. However, the kurtosis and skewness of the curve change according to the *minsup* value.

Table 5. *The number of k-length coding patterns with different minimum support values.*

Support	**Number of k-length patterns**							
(%)	*k=1*	*k=2*	*k=3*	*k=4*	*k=5*	*k=6*	*k=7*	*k=8*
5	10	50	147	234	225	135	39	6
6	10	47	108	139	125	54	10	1
7	10	43	82	100	68	26	3	0
8	10	40	71	77	42	11	1	0
9	10	39	65	52	20	5	0	0
10	10	36	51	38	17	3	0	0
11	9	30	40	33	11	1	0	0
12	9	25	33	21	8	1	0	0
13	9	25	27	16	4	0	0	0
14	9	22	23	12	2	0	0	0
15	9	20	20	10	1	0	0	0

C. COMPARISON RESULTS

In the last experiment, we compared four different sequential pattern mining algorithms in terms of running time, including PrefixSpan [1], SPADE [2], BIDE+ [3], and LAPIN [4]. This comparison is

important since the sequential pattern mining process is costly, especially when a low minimum support (*minsup*) value is given as input, or the dataset is dense and has long patterns. When handling large-scale data, some SPM algorithms have challenges and problems, including low processing speed, huge memory cost, and insufficient hard disk space [7]. The main reason behind this problem is the reduction in the elimination of candidate itemsets. Therefore, the number of frequent sequential patterns increases exponentially with respect to the *minsup*, instead of linearly. To increase the efficiency, an appropriate SPM algorithm should be chosen.

Figure 4 shows the comparison results when we set the minimum support threshold from 2% to 5% with an increment of 0.5%. It can be noted here that all the algorithms discover the same patterns but in different execution times. As expected, the runtimes decrease as the minimum support value increases. For example, the patterns were discovered in 2.20 sec. when *minsup* = 2, while they found in 0.76 sec. when *minsup* = 3. Therefore, the minimum support is important because it greatly impacts the execution time. The empirical results showed that the most efficient SPM algorithm is PrefixSpan in terms of computational time. For this reason, in this study, we used the PrefixSpan algorithm.

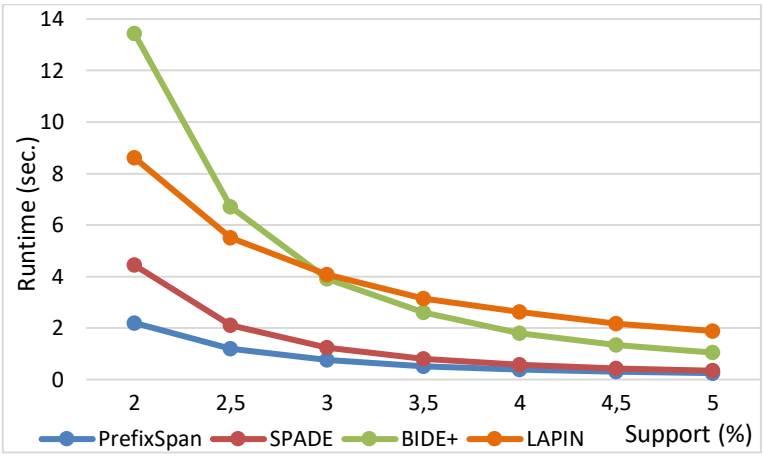

Figure 4. *Runtime (seconds) for retrieving frequent patterns*

Figure 4 shows the impact of the minimum support threshold on the running times of the algorithms. Furthermore, the *minsup* parameter also impacts the number of patterns as well as the interestingness of patterns found. As reported in [6], setting it to a very high value may lead to finding no frequent patterns, while a too-low value may generate many uninteresting and non-useful patterns. Moreover, the *minsup* parameter has an impact on the length of the patterns. As one decreases the minimum support, the number of longer patterns increases on high-dimensional datasets. Tuning the *minsup* parameter to an optimum value is a challenging task, especially for inexpert analysts. Deciding on the *minsup* value requires a trial-and-error process and therefore it is a tedious procedure for inexperienced users [6]. It may be noted here that if a hierarchical relationship between items is defined, different *minsup* values can be determined for different levels, instead of a uniform value. In multi-level sequential pattern mining [32], it is required to use higher *minsup* values at higher levels and reduced them at lower levels. Furthermore, it may be also stated here that the traditional SPM algorithms assume that each item has the same importance. The concept of multiple minimum supports (MMS) [33] extends the problem by allowing users to specify different *minsup* values for different items to reflect their own nature.

V. CONCLUSION AND FUTURE WORKS

This paper proposes a new framework, called *Source Code Miner* (SCodeMiner), which discovers frequent coding patterns within a software project. The proposed framework firstly transforms a source code into a sequence database and then applies a sequential pattern mining algorithm. This study is also original in that it compares four different algorithms in terms of computational time, including PrefixSpan, SPADE, BIDE+, and LAPIN. The experiments that carried out on an open-source software project showed that the proposed SCodeMiner framework is an effective mining tool in identifying coding patterns.

The main findings of the study can be concluded as follows:

- The empirical results showed that PrefixSpan outperformed the other alternative SPM algorithms (SPADE, BIDE+, and LAPIN) in terms of computational time. The results were obtained in a shorter time compared to the previous related studies [20-22] since it has been proven in [8, 30] that PrefixSpan is faster than PLWAP and GSP.

- With the proposed SCodeMiner framework, the general structure of the source code is extracted for general purposes. On the other hand, some previous approaches typically proposed for a specific purpose such as for analyzing software versions [5, 19, 23], usage patterns [20], software clones [18], crosscutting concerns [26], or software vulnerabilities [12, 13, 15, 17].

- The results showed that source codes usually contain frequent patterns of subroutine calls, mathematical operations, control flows, and loops such as <*LOOP MATH IF FUNC ENDIF ENDLOOP*>. On the other hand, when the source code of a graphic drawing tool is analyzed, the patterns can contain items related to drawing commands such as <*getGraphics setColor drawRect setcolor fillRect dispose*> as given in [20].

- The results showed that the number of frequent coding patterns decreased exponentially when the minimum support threshold increased. This behavior is also similar to the cases in [18, 20]. For this reason, an appropriate threshold value should be set to discover valuable frequent patterns.

- When the length of patterns was investigated, the frequency of results was distributed in the shape of the "bell" curve. The number of 3-length and 4-length patterns is usually the highest. However, the skewness and kurtosis of the curve can change according to the minimum support threshold value.

SCodeMiner has the potential to expand the application of data mining in the software engineering field, thanks to its advantages. Although found effective, it has several limitations. It is specially designed for the Java programming language; however, it is possible to extend the study for the other programming languages by only changing the tokenization step. Besides, it does not consider any user-defined constraint such as item constraint, time constraint, super-pattern constraint, or gap constraint. The patterns can be filtered according to a user-defined constraint. As a future study, we can use a closed, maximal, or high-utility SPM algorithm to present concise representations of coding patterns.

VI. REFERENCES

[1] A. Agrawal, M. Alenezi, R. Kumar, and R. A. Khan, "Securing web applications through a framework of source code analysis," *J. Comput. Sci.*, vol. 15, no. 12, pp. 1780-1794, 2019.

[2] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "An exploratory study on confusion in code reviews," *Empirical Software Eng.*, vol. 26, no. 12, pp. 1-48, 2021.

[3] S. Proksch, J. Lerch, and M. Mezini, “Intelligent code completion with Bayesian networks,” *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 1-31, 2015.

[4] M. M. Rahman, Y. Watanobe, K. Nakamura, and M. Bures, “A neural network based intelligent support model for program code completion,” *Sci. Program.*, vol. 2020, pp. 1-18, 2020.

[5] L. Kaur and A. Mishra, “Cognitive complexity as a quantifier of version to version Java-based source code change: An empirical probe,” *Inf. Softw. Technol*, vol. 106, pp. 31-48, 2019.

[6] A. A. Abdelaal, S. Abed, M. Al-Shayeji, and M. Allaho, “Customized frequent patterns mining algorithms for enhanced Top-Rank-K frequent pattern mining,” *Expert Syst. Appl.*, vol. 169, pp. 1-14, 2021.

[7] W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao, and P. S. Yu, “A survey of parallel sequential pattern mining,” *ACM Trans. Knowl. Discovery Data*, vol. 13, no. 3, pp. 1-34, 2019.

[8] J. Pei *et al.*, “Mining sequential patterns by pattern-growth: The PrefixSpan approach,” *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 10, pp. 1-17, 2004.

[9] M. J. Zaki, “SPADE: An efficient algorithm for mining frequent sequences,” *Mach. Learn.*, vol. 42, pp. 31-60, 2001.

[10] J. Wang and J. Han, “BIDE: Efficient mining of frequent closed sequences,” in *Proc. 20th Int. Conf. on Data Eng.*, Boston, MA, USA, 2004, pp. 79-90.

[11] Z. Yang, Y. Wang, and M. Kitsuregawa, “LAPIN: Effective sequential pattern mining algorithms by last position induction for dense databases,” in *12th Int. Conf. on Database Syst. for Adv. Appl.*, Bangkok, Thailand, 2007, pp. 1020-1023.

[12] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, pp. 1-11, 2021.

[13] S. Jeon and H. K. Kim, "AutoVAS: An automated vulnerability analysis system with a deep learning approach," *Comput. Secur.*, vol 106, pp. 1-24, 2021.

[14] C. D. Newman *et al.*, “On the generation, structure, and semantics of grammar patterns in source code identifiers,” *J. Syst. Softw.*, vol. 170, pp. 1-21, 2020.

[15] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Appl. Sci.*, vol. 10, pp. 1-16, 2020.

[16] Y. Ueda, T. Ishio, A. Ihara, and K. Matsumoto, “Mining source code improvement patterns from similar code review works,” in *IEEE 13th Int. Workshop on Softw. Clones*, Hangzhou, China, Mar. 2019, pp. 13–19.

[17] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology," *Plos One*, vol. 14, no 11, pp. 1-19, 2019.

[18] Y. Udagawa, “Maximal frequent sequence mining for finding software clones,” in *Proc. of the 18th Int. Conf. on Inf. Integration and Web-based Appl. and Services*, Singapore, Nov. 2016, pp. 26-33.

[19] H. Date, T. Ishio, M. Matsushita, and K. Inoue, “Analysis of coding patterns over software versions,” *Inf. Media Technol.*, vol. 10, no. 2, pp. 226–232, 2015.

[20] R. J. Akbar, T. Omori, and K. Maruyama, “Mining API usage patterns by applying method categorization to improve code completion,” *IEICE Trans. Inf. Syst.*, vol. E97.D, no. 5, pp. 1069–1083, May 2014.

[21] L. L. N. da Silva Junior, A. Plastino, and L. G. P. Murta, “What should I code now? ” *J. Univers. Comput. Sci.*, vol. 20, no. 5, pp. 797-821, 2014.

[22] H. Takei and H. Yamana, “IC-BIDE: Intensity constraint-based closed sequential pattern mining for coding pattern extraction” in *Proc. Int. Conf. on Adv. Inf. Networking and Appl.*, 2013, pp. 976-983.

[23] H. Date, T. Ishio, and K. Inoue, “Investigation of coding patterns over version history,” in *4th Int. Workshop on Empirical Softw. Eng. in Practice*, Osaka, Japan, 2012, pp. 40-45.

[24] H. Kagdi, M. L. Collard, and J. I. Maletic, “An approach to mining call-usage patterns with syntactic context,” in *ACM/IEEE Int. Conf. on Automated Softw. Eng.*, 2007, pp. 457-460.

[25] Y.-T. Kim, H.-T. Kong, and C.-S. Kim, “Analysis of characteristics and location of the appearance for codding pattern in the source code,” *J. Digit. Policy Manag.*, vol. 11, no. 7, pp. 165-171, 2013.

[26] T. Ishio, H. Date, T. Miyake, and K. Inoue, “Mining coding patterns to detect crosscutting concerns in Java programs,” in *Proc. Working Conf. on Reverse Eng.*, 2008, pp. 123–132.

[27] H. Tang, Y. Liu, and L. Wang, "A new algorithm of mining high utility sequential pattern in streaming data," *Int. J. Computational Intell. Syst.*, vol. 12, no. 1, pp. 342–350, 2019.

[28] I. Matloob, S. A. Khan, and H. U. Rahman, "Sequence mining and prediction-based healthcare fraud detection methodology," *IEEE Access*, vol. 8, pp. 143256-143273, 2020.

[29] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas, “A Survey of Sequential Pattern Mining,” *Data Sci. Pattern Recognit.*, vol. 1, no. 1, pp. 54-77, 2017.

[30] A. Palacios, A. Martinez, L. Sanchez, I. Couso, "Sequential pattern mining applied to aeroengine condition monitoring with uncertain health data," *Eng. Appl. Artif. Intell.*, vol. 44, pp. 10–24, 2015.

[31] P. Fournier-Viger *et al.*, “The SPMF open-source data mining library version 2,” in *European Conf. on Machine Learn. and Princ. and Practice of Knowl. Discovery in Databases*, 2016, pp. 36-40.

[32] S. Lianglei, L. Yun, and Y. Jiang, "Multi-level sequential pattern mining based on prime encoding," *Phys. Procedia*, vol. 24, pp. 1749-1756, 2012.

[33] Y.-H. Hu, F. Wu, and Y.-J. Liao, "An efficient tree-based algorithm for mining sequential patterns with multiple minimum supports," *J. Syst. Softw.*, vol. 86, pp. 1224-1238, 2013.