An Evaluation of Source Code Mining Techniques

Shaheen Khatoon
School of Computer and Applied Technology
Huazhong University of Science & Technology (HUST)
Wuhan, China

Azhar Mahmood
School of Computer and Applied Technology
Huazhong University of Science & Technology (HUST)
Wuhan, China

GUOHUI LI
School of Computer and Applied Technology
Huazhong University of Science & Technology (HUST)
Wuhan, PR China

***Abstract*— This paper reviews the tools and techniques which rely only on data mining methods to determine patterns from source code such as programming rules, copy paste code segments, and API usage. The work provides comparison and evaluation of the current state-of-the-art in source code mining techniques. Furthermore it identifies the essential strengths and weaknesses of individual tools and techniques to make an evaluation indicative of future potential.**

The pervious related works only focus on one specific pattern being mined such as special kind of bug detection. Thus, there is a need of multiple tools to test and find potential information from software which increase cost and time of development. Hence there is a strong need of tool which helps in developing quality software by automatically detecting different kind of bugs in one pass and also provides code reusability for the developers.

Keywords- Source code mining; literature review; Programming rule; Copy-paste code; API usage

I. Introduction

The primary goal of software development is to deliver high quality software in the least amount of time. To achieve these goals, Software Engineers are increasingly applying data mining algorithms to various software engineering tasks [1] to improve software productivity and quality.

To deliver high quality software, automatic bug detection remains one of the most active areas in software engineering research. Practitioners desire tools that would automatically detect bugs and flag the location of bugs in their current code base so they can fix these bugs. In this direction much work has been done to develop tools and techniques which analyze large amount of data about a software application such as source code, to uncover the dominant behavior or patterns and to flag variations from that behavior as possible bugs. One line of research in this direction is *Rule Mining Techniques* which induce set of rules from existing projects which can be used to improve subsequent development or new project development.

Another dominant work by mining source code is clone detection. Developers often reuse code fragments by copying and pasting (clone code) with or without minor adaptation to reduce programming efforts and shorten developing time. It also increase productivity since the code is previously tested and is less likely to have defects. However, clone code may cause potentially maintainability problem for example, when a cloned code fragment needs to be changed, for example change requirement or additional features, all fragments similar to it should be checked for the change. Moreover, the handling of duplicated code can be very problematic such as an error in one component is reproduced in every copy. This problem has focused the attention of researcher towards development of clone detection tools which allow developers to automatically find the locations in code that must be changed when related code segment changes.

Another line of related research is how to write APIs code. A software system interacts with third-party libraries through various APIs. Using these library APIs often needs to follow certain usage patterns. These patterns aid developers in addressing commonly faced programming problems such as what checks should precede or follow API calls, how to use a given set of APIs for a given task, or what API method sequence should be used to obtain one object from another.

In this paper, we provide a comprehensive comparison and evaluation of the currently available source code mining techniques and tools in the context of mining rules, detecting copy paste code and API usage. This work not only provides significant contributions to the source code mining research, but have also exposes how challenging it is to compare different tools, due to the diverse nature of the techniques and target languages. To date all the previous evaluation studies consider only one aspect of mining techniques such as clone detection or rules extraction and no comparative evaluation is available which detect various kind of patterns from source code in one pass. We aim to identify the essential strengths and weaknesses of individual tools and techniques to make an evaluation indicative of future potential e.g., when one aims to develop a new integrated or hybrid technique which address multiple challenges in one tool rather presenting another new tool.

The rest of this paper is organized as follows. After introducing some background of software mining in Section I, we provided a comprehensive literature review in section II. Section III presents an overall evaluation of source code mining tools and techniques in term of taxonomy. Section IV compares the existing techniques. Finally, Section IV concludes the paper and suggests directions for future work.

II. Related Work

A. *Mining rules from source code*

Rule mining techniques induce set of rules from existing projects which can be used to improve subsequent development or new project development. Several methods were proposed to detect rule-violating defects. Most of the studies used static source code analysis to find programming rules and subsequent rule violation as bugs. For example Engler et al., approach [2] and PR-Miner [3] mine function-pairing rules, CHRONICLER [4] mine function precedence protocols, Chang et al. [5] mine conditional rules and MUVI [6] mines variable-pairing rules

Engler et al.,[2]approach mines function pairing rules by using compiler extensions called *checkers* to match *rule templates*, Proposed tool extracts programming beliefs from acts at different location of source code by exploiting all possible paths between function call and cross check for violated beliefs . Since approach relies on developers to supply rule templates such as function A must be paired with function B and covers the given or explicit rules known in advance, it may miss many violations due to the existence of implicit rules. *PR-Miner* developed by Li and Zhou [3] find implicit programming rules and rule violations that is based on frequent item-set mining and does not require specification of rule templates. It can detect simple function pair-wise rules, complex rules as well as variable correlation rules. It computes the association in entire program elements by just counting the together occurrences of any two elements and not considering data flow or control flow which leads to increase number of false negative of violations in control path. CHRONICLER developed by Ramanathan et al.,[4] applies inter-procedural path-sensitive static analysis to automatically infer accurate function precedence protocols which specify ordering among function calls. CHRONICLER fundamentally differs from PR-Miner as it ensures path-sensitivity hence generate less number of false negative. Chang et al.,[5] proposed a new approach to mine implicit condition rules and to detect neglected conditions by applying frequent sub graph mining. . The approach requires the user to indicate minimal constraints on the context of the rules to be sought, rather than specific rule templates. However, frequent sub-graph mining algorithm does not handle directed graphs and multigraphs and require the modification leads to information loss so that precision is sacrificed in rule discovery. Another approach developed by Lu et al.,[6] called MUVI to mine variable pairing rules which applied the frequent itemset mining technique to automatically detect multi-variable inconsistent update bugs and multi-variable related concurrency bugs, which may result due to inconsistent update of correlated variables. Engler et al. [2] work also detect variable inconsistency through logical reasoning where as MUVI [6] detect inconsistencies using pattern analysis on multi-variable access correlations.

B. *Detecting copy paste code*

Several automated techniques for detecting code clones have been proposed differ by the level of comparison unit from single source lines to entire AST/PDG sub-trees/sub-graphs. However, we only focus on techniques which use data mining and few others leading techniques for clone detection such as CCFinder [7] and Dup [8] that use tokenization on the source code. Dup detect two types of matching code that is either exactly the same or name of parameters such as variable and constant are substituted. CCFinder detect clone code portions that have different syntax but have similar meaning and applies rule-based transformation such as regularization of identifiers, identification of structures, context information and parameter replacement of the sequence. Abstract syntax tree based approaches [9] and PGDs based [10] tools looks for sub trees and isomorphic graphs to find clones. In addition to above and many other technique we find only two approaches that, CP-Miner [11] and Clonedetection [9] which uses data mining to detect clones. CP-Miner uses frequent token sequence and flag bugs by recognizing deviations in mined patterns for renaming variables when copy-and-pasting the code. It transforms a basic block into number by tokenizing its component. Once all the components of a statement are tokenized, a hash value digest is computed using the “hashpjw” hash function. The ColSpan algorithm is applied to the resulting sequence database to find basic copy-pasted segments. By identifying abnormal mapping of identifiers among copy-paste segments, CP-Miner detects copy-paste related bugs, especially those bugs caused by the fact that the programmer forgot to modify identifiers consistently after copy-pasting. Whereas, Wahler et al. [9] approach find exact and parameterized clones at a more abstract level by converting the AST to XML by using frequent item set-mining technique. This tool first converts source code into Abstract Syntax Tree (AST) which contains complete information about source code by using parser. Frequent itemset mining algorithm inputs XML configuration file and find frequent consecutive statements. Proposed technique only finds exact and parameterized clones at a more abstract level.

C. *API Usage pattern*

Much research has been conducted to extract API usage rules or patterns from source code by proposing tools and approaches which helps developers to reuse existing frameworks and libraries more easily including [12-17]. In this direction, Michail, [14] described how data mining can be used to discover library reuse patterns in existing applications by developing a tool CodeWeb based on itemset and association-rule mining.

Prospector developed by Mandelin *et al.*, [13], automatically synthesize the list of candidate jungloid code based on simple query that described the required code in term of input and output . The Jungloid graph is created using both API method signatures and a corpus of sample client programs, and consists of chains of objects connected via method calls. Prospector mines signature graphs generated from API specifications and jungloid graphs. The retrieval is accomplished by traversing a set of paths (API method call sequences) from T_{in} to T_{out}. The code snippets returned by this traversal process are ranked using the length of the paths with the shortest path ranked first from Tin to Tout.

MAPO developed by Xie and Pei [17], mines frequent usage patterns of API through class inheritance. It uses API's usage history to identify methods call in the form of frequent subsequences. The code search engine receives a query that describes a method, class, or package for an API and then searches open source repositories for source files that are relevant to the query. The code analyzer analyzes the relevant source files and produces a set of method call sequences. The sequence preprocessor inline some call sequences into others based on caller-callee relationships and removes some irrelevant call sequences from the set of call sequences according to the given query. The frequent-sequence miner discovers frequent sequences from the preprocessed sequences. The frequent-sequence postprocessor reduces the set of frequent sequences in some ways.

Sahavechaphan and Claypool [15] developed, a context-sensitive code assistant tool XSnippet , that allows developers to query for relevant code snippets from a sample code repository to find code fragments relevant to the programming task at hand. A range of instantiation queries are invoked from java editor including *generic query* TQ_G that returns all possible code snippets for the instantiation of a type, to the *specialized type-based* TQ_T and *parent based queries* TQ_P, that return either type-relevant or parent-relevant results. User input the type of query, code context in which query is invoked and a specific code model instance to graph based Xsnippet system. Mining algorithm BFSMINE, a breath first mining algorithm traverses a code model instance and produces as output that represent the final code snippets meet the requirement of the specified query.

PARSEWeb developed by S. Thummalapenta, and T. Xie [16], uses Google code search for collecting relevant code snippets and mines the returned code snippets to find solution jungloids. The proposed technique described the desired code in the form of "Source $\rightarrow$ Destination" query which search for relevant code sample of source and destination object and download to form a local source code repository which is analyzed to constructs a directed acyclic graph. PARSEWeb identifies nodes that contain the given Source and Destination object types and extracts a Method-Invocation Sequences (MISs. PARSEWeb clusters similar MISs using a sequence postprocessor .The final MISs are sorts using several ranking heuristic and serves as a solution for the given query. PARSEWeb also uses an additional heuristic called query splitting that helps address the problem where code samples for the given query are split among different source files.

III. Taxonomy of Source Code Minning Techniques

This section encompasses the analysis on previously mentioned research contributions based on criteria that capture the main feature of each technique.

A supporting *tool* is developed by each approach as a plug-in for the programming environment. Source code is provided as input to tool and it applies data mining technique to detect frequently co-occurring patterns. Source code comprises of different elements such as functions, classes, variables, data types etc. Criterion *Input* shows which elements of source code are used as input by data mining tool. The criterion *output* indicate which type of mining information are extracted by tools developed by each approach e.g. programming rules, copy paste code, API usage. The criterion *Technique* entails the algorithm used by tool. Different algorithm used in source code mining research from data mining domain. Finally, criterion *Open Issues* indicates the research challenge not addressed by specific tool or technique. *Table 1* shows overall analysis of techniques and tools.

IV. Comparison of Source Code Mining Approaches

Both Engler et al., work and PR-Miner discover patterns involving set pairs of methods calls and functions, variables, data types that frequently appear in same methods and do not contain control structures or conditions among them, also the order of method calls is not considered. However, compared with Engler et al. work that extracts only function-pair based rules, PR-Miner extracts substantially more rules by extracting rules about variable correlations. Moreover, PR-Miner requires full parser to replace to work with other programming languages. CHRONICLER [4] which is fundamentally differs from PR-Miner as it ensures path-sensitivity hence generate less number of false negative as compare to PR miner. It differ from Engler et al., approach as it computes the precedence relationship based on program's control flow structure whereas Engler et al., approach detects relations between pairs of functions by exploiting all possible paths. MUVI [6] mines variable correlations and generate variable-pairing rules. Engler et al. [2] also detect variable inconsistency through logical reasoning where as MUVI [6] detect inconsistencies using pattern analysis on multi-variable access correlations.

Dup[8] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed Syntactically identical clones whereas CCFinder [7] applies additional transformations of source code that actually change the structure of the code, so that minor variations of the same syntactic form treated as similar. However, token-by-token matching is more expensive than line-by-line matching in terms of computing complexity since a single line is usually composed of several tokens. Dup, CCFinder and CloneDetection identify clone code that can be helpful in software amenability to identify section of code that should be replaced by procedure but do not detect copy paste related bugs. On the other hand CP - Miner [11] detect copy paste related bugs. Compared to CCFinder, CP-Miner is able to find 17.52% more copy-pasted segments because CP-Miner can tolerate statement insertions and modifications. whereas, Graph based analysis [10] can capture more complicated changes such as statement reordering, insertion and control replacement, compared with the common token-based approaches by capturing software's inherit logic relationship through PDG. Different mining techniques have been proposed in the literature to provide samples code which differs in the means that a developer uses to retrieve relevant examples from the repository, for example, Strathcona [18] use structural context to form a query is extracted automatically from the code a developer is writing. Xsnippets [15] uses class structure information such as parents, fields and methods of a class to define code context to query a sample repository for code snippets relevant to the object instantiation task at hand. Prospector [13] , Parseweb [16] and MAPO [17] defines a query that describes the desired code.

TABLE I. TAXONOMY OF SOURCE CODE MINING TOOL AND TECHNIQUES

	Description	**Tool**				**Open Issues**	**Ref.**
		Name	***Input***	***Output***	***Technique***		
Rule- Mining	Need rules to check against program code by inferring code believes and cross check for contradiction	Static Analyzer	Functions	Pair-wise programming rules.	Statistical analysis	Fixed rule templates, only identify pair wise programming rules	[2]
	Frequent itemset mining for pair-wise, multi-functions and variable correlation rules	PR-Miner	Functions, variable and data type	Pair-wise, complex and variable correlation rules	Item-set mining	Does not consider inter-procedural analysis, data flow and control relationship	[3]
	Frequent subsequence mining to infer function precedence protocols	CHRONICLER	Functions	Function calls ordering rules	Frequent subsequence mining	Does not take account of data flow or data dependence	[4]
	Graph based mining to search conditional rules	Framework	Program Dependence Graphs	Graph minor as conditional rules	Frequent item-set and sub-graph mining algorithm	Require manual inspection for valid rules that may miss some instances of rules during inspection.	[5]
	Frequent itemset mining to extract variable correlations	MUVI	Functions, Global, class & structural variables	Variable pairing rules	Frequent item-set mining	Only handled variable access directly by caller functions	[6]
Detecting copy paste code	Suffix trees for tokens per line	Dup	Sequence of lines	Line by line clones	Suffix tree based matching	Does not detect clone code portions having different syntax but similar meaning.	[8]
	Token normalizations, then suffix-tree based search	CC-Finder	Sequence of tokens	Clone pairs	Token comparison Suffix tree based matching	Does not detect changes such as statement reordering, insertion and control replacement.	[7]
	Data mining for frequent token sequences	CP-Miner	Statement sequence	Copy-paste code	frequent subsequence & tokenization	Same syntax but different semantic are detected as copy paste segments	[11]
	XML representation of ASTs with frequent itemsets techniques of data mining	Clone-Detetion	XML representation of ASTs	Clone pairs.	Frequent item set mining	It does not detect complicated changes i.e. statement reordering, insertion and control replacement.	[9]
	Searching similar sub graphs in PDGs	Framework	PDG	Matching sub-graph	Spatial search & Graph matching	Limitation in search speed and pattern accuracy	[10]
API Usage	Discover library reuse patterns using association rule mining	CodeWeb	Components, classes, and functions	Library reuse pattern through class inheritance	Item-set and association-rule mining	To use CodeWeb developer must find similar applications of interest in advance.	[14]
	Context based matching of related source code from example repository	Strathcona	Structural context of code	List of relevant code under development	Heuristic matching	Each heuristic is generic, not specific to a particular task of object instantiation	[18]
	Mining past repositories to search for a call chain that has previously been used.	Prospector	API Method signature/Class type	API Jungloids	Signature graph matching	It returns many irrelevant examples or in some cases too few qualified examples	[13]
	Context sensitive code assistant to mining sample code repository for relevant code	XSnippet	Inheritance hierarchy, fields and methods	API code snippets	Graph mining	XSnippet is limited to the queries of a specific set of frameworks or libraries.	[15]
	Mines API usage history to identify call patterns	MAPO	Method, class or package	sequencing information among method calls	Frequent sequence mining	It does not synthesized code fragments from mined frequent can be directly inserted into developers' code.	[17]
	Search web for related code and mine the return code to find solution	ParseWeb	Objects	Method Invocation sequence (MIS)	Clustering	It only suggests the frequent MISs and code samples cannot directly generate compliable code.	[16]

V. Conclusion

In this paper we have provided concise but comprehensive survey of three types of source code mining tools and techniques such as mining rules, copy-paste code and API usage. So far this is the first survey which includes combination of different techniques .Comparison of techniques and tools shows there is a no single tool which is superior to all other in all aspects because all tools have strength and weaknesses and intended for different task and context. However, a combination of these three source code mining techniques help one to understand how to design a hybrid/integrated technique to be robust across all types of software patterns that can help bug detection as well as help developers to write relevant API code. The comparison also helps how to employ a set of different tools to achieve better results.

In future we are going to develop an integrated framework which can automatically find all the patterns from source code in one pass and suggest developer potential bug locations for quality software development and relevant code suggestion for rapid software development.

References

[1] A. Hassan, and T. Xie, “Mining software engineering data,” in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, pp. 503-504.

[2] D. Engler, D. Chen, S. Hallem *et al.*, “Bugs as deviant behavior: A general approach to inferring errors in systems code,” *ACM SIGOPS Operating Systems Review,* vol. 35, no. 5, pp. 57-72, 2001.

[3] Z. Li, and Y. Zhou, “PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code,” in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005, pp. 306-315.

[4] M. Ramanathan, A. Grama, and S. Jagannathan, “Path-sensitive inference of function precedence protocols,” in 29th International Conference on Software Engineering (ICSE 2007), 2007, pp. 240-250.

[5] R. Chang, A. Podgurski, and J. Yang, “Finding what's not there: a new approach to revealing neglected conditions in software,” in Proceedings of the 2007 international symposium on Software testing and analysis, 2007, pp. 163-173.

[6] S. Lu, S. Park, C. Hu *et al.*, “MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs,” *ACM SIGOPS Operating Systems Review,* vol. 41, no. 6, pp. 103-116, 2007.

[7] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, pp. 654-670, 2002.

[8] B. Baker, “On finding duplication and near-duplication in large software systems,” in Proc. Second IEEE Working Conf. Reverse Eng., 1995, pp. 86-95.

[9] V. Wahler, D. Seipel, J. Wolff *et al.*, “Clone detection in source code by frequent itemset techniques,” in Fourth IEEE International Workshop on Source Code Analysis and Manipulation, 2004, pp. 128-135.

[10] W. Qu, Y. Jia, and M. Jiang, “Pattern mining of cloned codes in software systems,” *Information Sciences*, 2010, 2010.

[11] Z. Li, S. Lu, S. Myagmar *et al.*, “CP-Miner: A tool for finding copy-paste and related bugs in operating system code,” in Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6, 2004, pp. 20.

[12] M. Acharya, T. Xie, J. Pei *et al.*, “Mining API patterns as partial orders from source code: from usage scenarios to specifications,” in Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007, pp. 25-34.

[13] D. Mandelin, L. Xu, R. Bodík *et al.*, “Jungloid mining: helping to navigate the API jungle,” *ACM SIGPLAN Notices,* vol. 40, no. 6, pp. 48-61, 2005.

[14] A. Michail, “Data mining library reuse patterns using generalized association rules,” in Proceedings of 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland, 2000, pp. 167-176.

[15] N. Sahavechaphan, and K. Claypool, “XSnippet: mining for sample code,” *ACM SIGPLAN Notices,* vol. 41, no. 10, pp. 413-430, 2006.

[16] S. Thummalapenta, and T. Xie, “Parseweb: a programmer assistant for reusing open source code on the web,” in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 204-213.

[17] T. Xie, and J. Pei, “MAPO: Mining API usages from open source repositories,” in Proceedings of the 2006 international workshop on Mining software repositories, 2006, pp. 54-57.

[18] R. Holmes, and G. C. Murphy, “Using structural context to recommend source code examples,” in Proceedings of the 27th international conference on Software engineering, 2005, pp. 117-125.