Comparison and evaluation of source code mining tools and techniques: A qualitative approach

Article *in* Intelligent Data Analysis · May 2013

DOI: 10.3233/IDA-130589

CITATIONS
2

READS
197

3 authors, including:

Shaheen Khatoon
Huazhong University of Science and Technology
23 PUBLICATIONS **59** CITATIONS
SEE PROFILE

Mahmood Azhar
Huazhong University of Science and Technology
23 PUBLICATIONS **53** CITATIONS
SEE PROFILE

Intelligent Data Analysis 17 (2012)
IOS Press

Comparison and Evaluation of Source Code Mining Tools and Techniques: A Qualitative Approach

Shaheen Khatoon [a], Guohui Li [a,*] , Azhar Mahmood [a]

[a] *Huazhong University of Science & Technology (HUST), Wuhan, PR China*

Abstract. Program source code substantially is structured and contains semantically rich programming constructs such as variables, functions, data structures, and program structures which indicate patterns. Mining source code by using different data mining techniques to extract the valuable hidden patterns is the new revolution in software engineering. Over last decade many tools and techniques have been proposed by researcher to extract pertinent information and uncover relationships and trends from source code about a particular characteristic of Software Engineering (SE) tasks. These efforts have resulted in wide range of research body but currently there is no comprehensive overview exists.

This paper surveys the tools and techniques which rely only on data mining methods to determine patterns from source code in context of programming, bug detection, maintenance, program understanding and software reuse. The work provides comparison and evaluation of the current state-of-the-art source code mining tools and techniques, and organizes the large amount of information into a coherent conceptual way. Thus the survey provides researchers with a concise overview of source code mining techniques and assists practitioners the selection of appropriate techniques for their work.

The result of this review shows existing studies focus on one specific pattern being mined from source code 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 and generates relevant API code automatically to help in decreasing overall software development time.

Keywords**:** Source code mining; Data mining; patterns; Programming rule; Copy-paste code; Bug detection; API usage.

1. Introduction

Data mining focuses on the techniques for non-trivial extraction of implicit, previously unknown and potentially useful information from very large amounts of data [1-4]. In relation to this, a large amount of SE data is also produced in software development process such as requirement document, design specifications, source code files and information about bugs. These sources contain a wealth of valuable information.

In recent years, software practitioners and researchers have recognized that valuable information can be extracted from SE data to support software development practices and software engineering research. In this direction Mining Software Repositories (MSR) is specially promoted because of commonly availability of software version control repositories, bug tracking repositories and archived communications for most software projects. Practitioners are increasingly applying data mining techniques

* Corresponding Author. E-mail : guohuiliwh@gmail.com

on MSR to support software development practice and various SE tasks[5].

Software artifact such as source code is an important source of data that can be mined for interested patterns. It is typically structured and also contains semantically rich programming constructs such as variables, functions, data structures, and program structures which indicate patterns. We can uncover useful and important patterns and information by mining source code. Various data mining applications in software engineering have employed source code to aid software maintenance, program comprehension and software components' analysis.

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 looking for tools which automatically detect different type of bugs to deliver high quality software and want to reuse existing frameworks or libraries for rapid software development. To accomplish these tasks practitioners increasingly applying data mining algorithms to various software engineering data [6] to improve software quality and productivity. To deliver high quality software automatic detection of bugs 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 source code data, to uncover the dominant behavior or patterns and to flag variations from that behavior as possible bugs. One major area in this direction is Rule Mining Techniques which induces set of rules from source code of existing projects and anomalies are uncover by looking for violation of specific rule. Most of the studies used static source code analysis to find programming rules and subsequent rule violation as bugs [7-11].

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 the development 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 in case of change requirement or additional features, all fragments similar to it should be checked for that 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. 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[12-17]. Here we focus only on techniques which are using data mining methods and few others leading techniques for clone detection.

The reuse of both software library and application framework is an important activity for rapid software development at the source level. In recent development setting programmer relies on frameworks and libraries, such as C++ libraries, Java packages, Eclipse packages, and in-house libraries [18-21] that

privilege the programmer to create high quality, full featured applications on-time. However, due to rapid change in software these libraries are not well documented and having complex APIs. By mining API usage patterns one can identify dominant and correct library usage patterns across many projects. Different mining techniques have been proposed in the literature which provide samples code [20], [22-25]. The techniques are different in the way how the developer queries the target source repository to retrieve relevant code example.

Data mining can also be applied to source code change histories to predict bugs and locate possible changes. Many new bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly. Studies in [26-37] have been conducted to search for possible bugs and guide software changes by mining source code histories.

Source code also plays an important role in understanding large system because the documentation for these system rarely exists and if exists is not up-to-date. Practitioners also apply data mining techniques on large software system for program comprehension, architecture recovery or some other tasks mentioned in section 3.

In this paper, we provide a comprehensive comparison and evaluation of the state of art static source code mining techniques. To date little effort has been spent to evaluate on this leading area of research. Apart from our own initial short survey [38] , which examined three approaches, two other surveys have been presented. Kagdi et al. [39] provided survey of approaches those uses frequent-pattern mining to mine software repositories for various software evolution tasks. The surveyed approaches require extensive history of software revisions in repository to be effective. In the other work, Halkidi et al. [40] surveyed approaches those applying data mining techniques on various sources of software engineering data. In contrast, this work focus on a survey approaches those examine the relationship between source code entities, change relationship or reuse of component. This type of investigation is research on analysis methods to support testing, programming and maintenance task.

This work not only provides significant contributions to the source code mining research, but have also exposes how challenging to compare different tools due to the diverse nature of the techniques and target languages.

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. Moreover, by this survey we have made available prominent tools and techniques pertaining to source code mining to practitioners who are interested to improve software development process or its related challenges.

The rest of this paper is organized as follows. After introducing some background of software mining in Section 2, we provide a comprehensive overview of existing techniques on mining source code data in section 3. Section 4 presents taxonomy of source code mining tools and techniques in term of general criteria in form of table. The organization of surveyed approaches in term of data mining approach is

presented in section 5. Comparison of source code mining tools in term of several evaluative criteria is presented in section 6. Critiques on evaluated tools in term of strength and weaknesses are presented in section 7. Finally, discussion and future trends are highlighted in section 8 and the paper is concluded in section 9.

2. Software mining

Software mining encompasses the use of data mining to improve software productivity and quality. We speak of software mining when Software Engineering (SE) data is used as input to data mining algorithm to extract latent valuable information from SE data. There are various sources of software engineering data such as code bases, execution traces, historical code changes, mailing lists, software metrics or bug databases on which data mining can be applied to support various SE tasks.

We specifically speak of source code mining when software engineering data pertaining to static source code is used as input to data mining technique. Static analysis of source code has several well-known benefits. Examining the source code without actually executing the code makes the quality of test suites, an active area of research [41, 42]. Static analysis also allows code to be tested that is difficult to run in all environments, such as device drivers.

2.1 General code mining process

This section discusses an overall summary of the code mining process discussed in paper. The overall data mining process applied on code bases is shown in Fig 1. The overall code mining process included more or less following steps:

- *Collecting target data.* Collecting source code to local repository and determine what type of dataset to mine and what type of SE tasks can be assisted by mining.
- *Preprocessing.* It include extracting relevant data such as static methods, call sequences from source code and removing uninteresting elements which cause noise. Moreover, including them in the data set would significantly increase the computation of mining algorithms.
- *Code transformation.* Transforming data in a way adoptable to particular data mining algorithm. For example input format for frequent itemset mining is itemset database where each function is an itemset. To accomplish this task we have to replace each element of method call with a distinct number in the itemset database being fed to mining algorithm.
- *Mining techniques.* Choosing the appropriate mining algorithm to perform the desired function to find the pattern in data. Various data mining functions such as clustering, classification, association discovery, pattern mining and pattern matching are used to mine SE data.
- *Post processing.* It transforms mining results into appropriate format required to assist SE task. For example in the preprocessing step, each distinct method call replaces with set of numbers in itemset

database being fed to the mining algorithm. In this step numbers are converted back into distinct method call.

- *Ranking.* Candidate results produced by mining algorithm are too many and often irrelevant. Ranking applies manual analysis or automated heuristics to separate valid patterns from coincidence or uninteresting patterns.

Software patterns mined by source code mining process can be used in following different ways.

- They can be stored in a specification database so that programmers can refer in future project development to improve the efficiency of software systems
- Frequently appeared patterns can be used to discover implicit programming rules
- The mined patterns can be used to discover related defects.
- Relevant patterns can be used for code optimization
- Pattern mining could help to determine code reusability

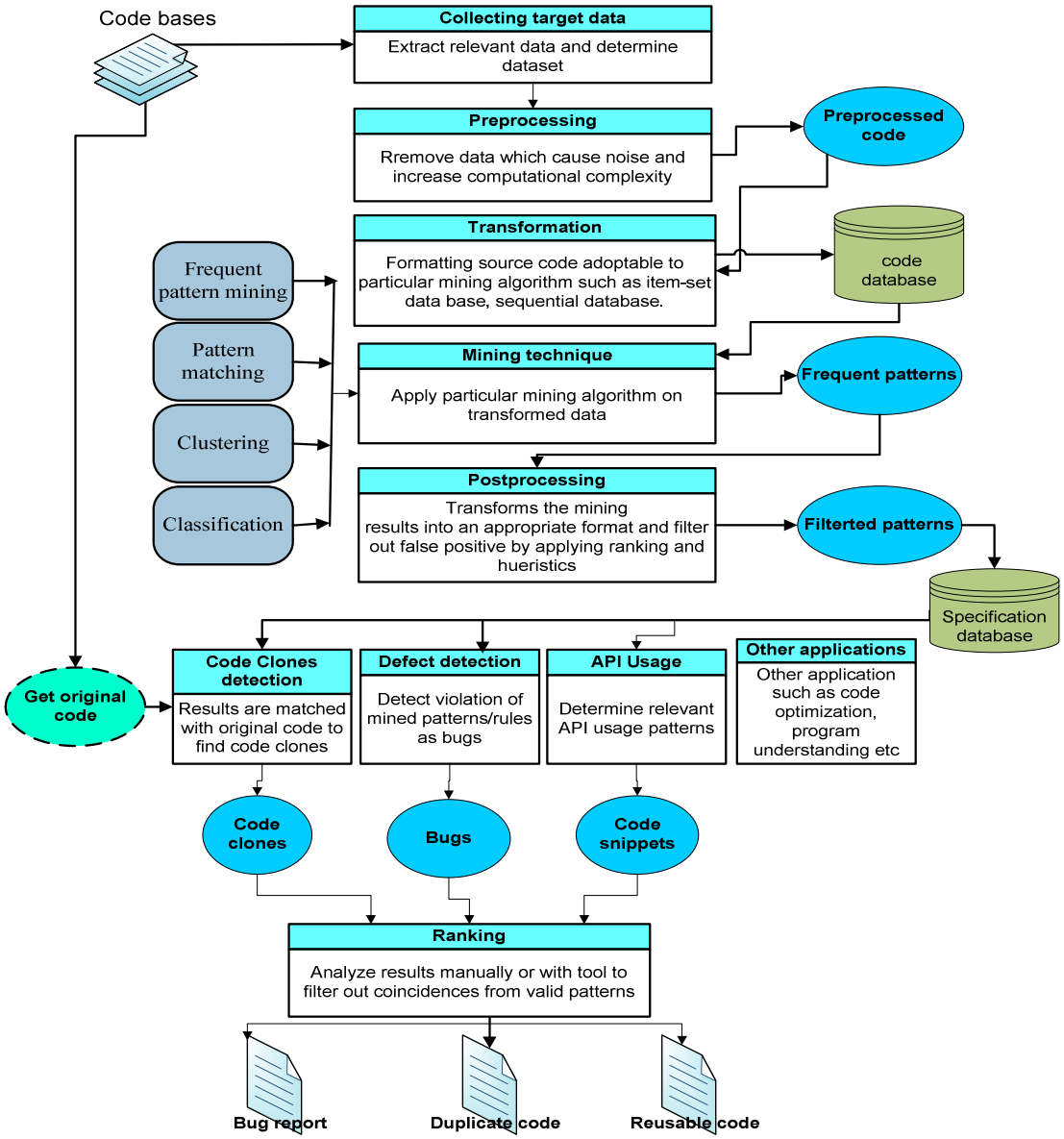

Fig.1. Code mining process

3. Overview of static source code mining tools and techniques

Various approaches have been developed to benefit software engineering tasks by using data mining that deal with different types of SE data. The main contribution of this work is to investigate how data mining techniques used on source code data to improve software quality and productivity. This survey is based upon the nature of information mined from target source code which subsequently used to improve software development. For example to reveal underlying correlation among the data set, one may go for mining such rules which show some association between data element. These rules subsequently can be

used to detect software bugs as a possible rule violation. Clone mining techniques helps Software engineers to find duplicate code which can subsequently used in maintenance phases, code optimization etc. First we focus on those approaches that are relying on rule mining to detect software anomalies, data mining methods to find clone code and relevant API usage patterns as well as mining version histories to detect bugs and change prediction. Moreover, we also highlight on other areas that use data mining approaches on source code data. Hence, by reading this survey paper researchers are directed to specific source code activity they want to do in future. This list is by no means exhaustive but represents a number of different prominent investigations.

3.1 Mining rules to detect bugs

Rule mining techniques induce set of rules from existing projects which can be used to uncover potential bugs as violation of specific program rule. Several methods were proposed to mine program source code and detect software anomalies as a possible rule violation.

Engler et al. [7] developed a static verification tool by using compiler extensions called checkers (written in the Metal language) to match rule templates, derived from knowledge of typical programming errors, against a code base. 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 e.g. a dereference of a pointer, p, implies a belief that p is non-null, a call to "unlock (1)" implies that 1 was locked etc. Rule template represent general programming rules such as such as “<a> must be paired with <b>” and Checkers, match rule templates to find the rules instance and discover code locations where it violates a rule that match an existing template. Two types of rules categories: MUST-rules (inferred from acts that imply beliefs code “must” have) and MAY-rules (Inferred from acts that imply beliefs code “may” have) are identified. For MUST rules internal consistency is checked and contradictions is directly flagged as bugs; for MAY-rules, a statistically based method is used to identify whether a possible rule must hold. Proposed approach applies statistical analysis, based on how many times the rule holds and how many it does not to rank deviations from programmer beliefs inferred from source code.

PR-Miner (Programming Rule Miner) [8] uses item-set mining to automatically extract general programming rules from software code written in an industrial programming language such as C and detect violations. It transforms a function definition into an item-set by hashing program elements to numbers. In this conversion process, similar program elements are mapped to the same number, which is accomplished by treating identifiers with the same data types as identical elements, regardless of their actual names. By using the frequent item-set mining algorithm called FPclose, PR-Miner extracts rules from possible combination of multiple program elements of different types including functions, variables, data types, etc. that are frequently used together and find association among them. For efficiency PR-Miner generates only closed rules from a mined pattern. The rules extracted by PR-Miner are in general forms, including both simple pair-wise rules and complex ones with multiple elements of different types. By identifying which

elements are used together frequently in the source code, such correlated elements can be considered a programming rule with relatively high confidence.

CHRONICLER [9] applies inter-procedural path-sensitive static analysis to automatically infer accurate function precedence protocols which specify ordering among function calls e.g. A call to *pthread_mutex_init* must always be present on program paths before a call to *pthread_mutex_lock*. Precedence relationship is computed using program's control-flow structure and stored into a repository which analyses using sequence mining techniques to generate a collection of feasible precedence protocols. CHRONICLER first generates the control flow graph for each procedure and reverses the direction of all edges in control-flow graph to construct the precedence relation. The graphs obtained are fed into the relation builder and a cycle of relation, constraint, and constraint summary calculations is executed. The sequences obtained as a result of this process are then fed to a sequence mining tool. MAFIA [43] to generate the item sets that appear frequently based on a given confidence threshold. The protocols output by the sequence miner and the associated violations are ranked by processing them according to the confidence, length and frequency of occurrence of the protocol. The output of the entire process is a ranked order of function precedence protocols. Deviations from these protocols found in the program are tagged as violations and also represent potential sources of bugs.

Some complex rules may indicate variable correlations, i.e. these variables should be accessed together or modified in a consistent manner. In this direction Lu et al. [11] developed a tool called MUVI to mine variable pairing rules which applied the frequent itemset mining technique to automatically detect two types of bug i.e. (1) multi-variable inconsistent update bugs and (2) multi-variable related concurrency bugs, which may result due to inconsistent update of correlated variables, the variables that need to be accessed together. For example "*thd->db_length*" describes the length of the string "*thd->db*", so whenever "*thd->db*" is updated, "*thd->db_length*" should be updated consistently. The "access together" variables are those which appear in the same function with less than maximum distance statement apart, and collected by statically analysis of each function to form *Acc-Set*. MUVI's applied FPclose algorithm to *Acc_Set* database, consisting of the *Acc_Sets* of all functions from the target program and output set of variable accessed more than minimum support number of functions. MUVI only focused on two kinds of variables: global variables and structure/class fields.

A new approach to mine implicit conditional rules is proposed by Chang et al. [10]. The proposed work detects neglected conditions by applying frequent sub graph mining on C code. Neglected conditions are missing conditions, cases or path which if not carefully revealed software open to many security vulnerabilities. Software open to security vulnerabilities often exploited by the attackers such as buffer overflow, SQL injection, Cross-site scripting, format string attacks. In proposed approach program units are represented as Program Dependency Graphs (PDGs) generated by CodeSurfer a static analysis tool. The PDGs are enhanced by adding directed edges, called shared data dependence edges path. The resulting graphs are called enhanced PDGs (EPDGs). Potential rules are represented by graph minors by contracting

some paths edges. Because EPDG minors represent transitive (direct and indirect) intra-procedural dependences between program statements, they capture essential constraints between rules element. Rules are modeled as graph minors of enhanced procedure dependence graphs (EPDGs), in which control and data dependence edges are augmented by edges representing shared data dependences. The next step is to mine the resulting EPDGs to identify candidate rules. Rules are found by mining a database of near transitive closure of EPDGs, using frequent sub-graph mining algorithm, to find recurring graph minors on the assumption that the more a programming pattern is used, the more likely it is to be a valid rule. After valid rules are discovered and confirmed the graph database is searched again using a heuristic graph matching algorithm, to find rule violations corresponding to neglected conditions.

Kagdi et al. [44] presented two approaches for mining call usage patterns from source code. The first approach is based on the idea of itemset mining. It identifies frequent subsets of items that satisfy at least a user defined minimum support. As a result unordered patterns related to functions calls are generated. Sequential pattern mining applies on partial ordered list of function calls that produces more accurate results and less number of false positive. In general terms these approaches can assist with mining patterns of call usage and thus identifying potential bugs in a software system.

3.2 Mining code clones patterns

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development for quick performance gains during development and maintenance. Studies shows that 20-50% of large software systems contain so called clones“ similar program structures” repeated many times within or across programs in variant forms [12, 45, 46]. Independently of the reasons why they arise, such cloned structures hinder future maintenance. They complicate programs, make it difficult to trace the impact of change and increase the risk of update anomalies. Other problems triggered by clones include replication of unknown bugs, code bloat and dead code [12]. Many software engineering tasks such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining (clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments which makes clone detection an important and valuable part of software analysis[47]. There is a multitude of techniques for detecting code clones have been proposed in literature. Clone detectors [12, 13, 15-17, 46, 48-55] identify similar code automatically. Each technique is differ by the level of comparison unit of source code such as parameterized tokens strings [12, 13] AST [14, 15, 50, 54] and PGDs [16, 17]. Here we focus on work which used data mining and few other basic tools in this area.

A string based approach to locate code duplication is proposed by Baker [12].It uses sequence of lines as a representation of source code and detects line-by-line clones. A tool called Dup is developed which detects

two type of matching code that is either exactly the same or name of parameters such as variable and constant are substituted. It performs the following sub processes: 1) Lines of source files are first divided into tokens by a lexical analyzer, 2) replacement of tokens (identifiers of functions, variables, and types) into a parameter identifier, 3) parameter tokens are encoded using a position index for their occurrence in the line. 4) All prefixes of the resulting sequence of symbols are then represented by a suffix tree, a tree where suffixes share the same set of edges if they have a common prefix. 5) Extraction of matches by a suffix-tree algorithm, if two suffixes have a common prefix, clearly the prefix occurs more than once and can be considered a clone.

CCFinder [13] is another token based clone detection technique with additional transformation rules to remove minor difference in source code. It transforms source code into tokens sequence through lexical analyzer to detect clone code portions that have different syntax but have similar meaning. It also applies rule-based transformation such as regularization of identifiers, identification of structures, context information and parameter replacement of the sequence. Source normalizations is used to remove superficial differences such as changes in statement bracketing (e.g. if (a) b=2; vs. if (a) {b=2;}). Finally, clone pairs, i.e., equivalent substrings in the token sequence, are identified using suffix-tree matching algorithm.

CP-Miner [53] applies data mining to identify copy-paste defect in operating system code. By using frequent subsequence mining and tokenization technique it detects copy-paste-related wrong variable-name bugs. It transforms a basic block into number by tokenizing its component such as variable, operators, constants, functions etc. Once all the components of a statement are tokenized, a hash value digest is computed using the *hashpjw* hash function. As a result program become a large sequence which is broken into small sequence by choosing basic block as a unit to get each basic block as a sequence. The CloSpan 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.

Another clone detection tool is CloneDetection proposed by Wahler et al. [15]. This tool first converts source code into Abstract Syntax Tree (AST) which contains complete information about source code by using parser. AST is than transformed into XML representation. XML files are further configured to define metadata to show how statements are nested and considered as clones. 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.

Qu et al. [17] Proposed a framework for pattern mining of clone code using spatial and graph base analysis of source code. Source code is first transformed into Program Dependency Graph (PDG), each line of code is encoded to a hash value by converting each node and edge of PDG into index string by applying hashing algorithm. Hence, original source code is converted into encoded graphic sequence. Spatial search is

performed on encoded graphic sequence by using Winnowing algorithm to find pair wise matches which serves as input to graph based pattern mining. For graph base pattern mining PDG for each line pair is retrieved and graph based pattern matching VF algorithm applied to find matched sub graph inside the PDG pair. False positive pruning and pattern composition techniques are used to update the pattern database and discover more accurate and meaningful patterns of cloned codes. Finally the detected software patterns can be used for various applications such as pattern analysis, related defect discovery and code optimization.

Rysselberghe and Demeyer [55] present a technique to identify frequently applied changes by mining version histories based on clone detection. CVS deltas are examined and their corresponding source code changes are recorded in a text file. A clone-detection tool CCFinder using parameterized token matching is applied to this text file to find similar pairs of source code changes i.e. clones. The CVS deltas corresponding to these clones are considered as the Frequently Applied Changes (FACs). It is observed these FACs are typically caused by a 'well-established' solution at one place being replicated at other locations (later eliminated by a function) moving code (considered deleted and then added) and temporary addition of code that was later deleted. These changes are then studied to identify possible maintenance activities, such as refactoring. They also proposed matching frequently applied changes to bug reports helping to identify bugs in the code and solutions to these bugs. The technique is evaluated on the three-year version history of an open-source system, Tomcat. Both high and low threshold values on the number of matching tokens are experimented to detect FACs.

Basit and Jarzabek [54] introduced the concept of structure clone and proposed a tool called *Clone Miner*. Proposed tool first extracts simple clones form the source code (similar code fragments) by using simple clone detector, as a default front-end tool [56]. Then by using frequent closed item sets mining to detect recurring groups of simple clones in different method and files. File clone sets and method clone sets are found by the process of clustering from the lower level structural clones respectively. Using this mechanism highly similar group of files and methods are found which consist of groups of cloned entities at successively higher levels of abstraction. Evaluation of proposed technique on several case studies shows the procedure of detecting structural clones can assist with understanding the design of the system for better maintenance and reengineering.

3.3 Mining API usage pattern

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. Much research has been conducted to extract API usage rules or patterns from source code by proposing tools and approaches that help developers to reuse existing

frameworks and libraries more easily including [22-25, 57, 58].

In this direction Michail [58] developed a tool named CodeWeb which described how data mining can be used to discover library reuse patterns in existing applications. It mines association rules such as what application classes inheriting from a particular library class often instantiate another class or one of its descendants. Based on itemset and association-rule mining CodeWeb uncover entities such as components, classes and functions that occur frequently together in library usages. Michail explains by browsing generalized association rules, a developer can discover usage patterns in a way which takes into account inheritance relationship.

Holmes et al. [20] have developed Strathcona, an Eclipse plug-in, that enables localization of relevant code in an example repository. Their approach is based on six heuristics that match the structural context descriptions (parents, invocations and types) encapsulated in the developer code with that encapsulated in the example code. Each heuristic is used to query the code repository, returning a set of methods and classes where the result context matches the query's context. The result is a set of examples (source code examples) that occur most frequently when applying all heuristics.

Mandelin et al. [23], developed a tool called Prospector for automatically synthesize the list of candidate jungloid code based on simple query that described the required code in term of input and output . A Jungloids is a simple unary expression which helps to determine a possible call chain between a source type and a target type. A jungloid query is a pair (T_{in} , T_{out}) where T_{in} and T_{out} are source and target object types respectively. 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} where each path represents a code fragment and a set of paths in turn composes all code fragments to form a code snippet. The code snippets returned by this traversal process are ranked using the length of the paths with the shortest path ranked first from T_{in} to T_{out}.

Sahavechaphan and Claypool [22] developed a context-sensitive code assistant tool XSnippet , an Eclipse plug-in 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. XSnippet extends Prospector and adds additional queries, ranking heuristics and mining algorithms to query a code snippet from a sample code repository for code snippets relevant to the object instantiation at hand. XSnippets [22] transforms source classes into corresponding source code model instances using directed acyclic graph which captures class structure represented by inheritance hierarchy, fields, method and class behavior. Code relevance is defined by the context of the code, both in terms of the parents of the class under development as well as lexically visible types for a given method contained in class. A range of instantiation queries are invoked from java editor including generic query TQG that returns all possible code snippets for the instantiation of a type, to the specialized type-based TQT and parent based queries TQP, 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, a set of paths that represent the final code snippets meet the requirement of the specified query. Paths can be either within the method scope or outside of the method boundaries, ensuring that relevant code snippets that are spread across methods are discovered. Ranking heuristics are applied to resultant code to remove duplicate, non compilable and non executable path and rank the code on the basis of context, frequency and length of snippet. The pruned mining paths are passed to the Snippet Formulation process that transforms each path to corresponding code snippet.

MAPO [25] developed by Xie and Pei, 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 returned by the code search engine and produces a set of method *call* sequences, each of which is a *callee* sequence for a method defined in the source files. The sequence preprocessor inlines 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.

PARSEWeb [24] developed by Thummalapenta and Xie uses Google code search for collecting relevant code snippets dynamically and mines the returned code snippets to find solution jungloids. The proposed technique is based upon the simple query which described the desired code in the form of "Source $\rightarrow$ Destination" which search for relevant code sample of source and destination object usage and download to form a local source code repository. PARSEWeb analyzes the local source code repository and constructs a Directed Acyclic Graph (DAG). By searching the nodes in DAG PARSEWeb identifies nodes that contain the given Source and Destination object types and extracts a Method-Invocation Sequences (MISs) that can transform an object of source type to object of destination type by calculating the shortest path between nodes. Similar MISs are clustered using a sequence postprocessor to form solution for given query .The final MISs are sorts using several ranking heuristic and serves as a solution for the given query. The suggested MIS contains all necessary information for the programmer to write code for getting the Destination object from the given Source object. Parseweb also suggest the relevant code sample as well as MISs and uses an additional heuristic called query splitting that helps to address the problem where code samples for the given query are split among different source files.

3.4 Mining co-changes and bug fix changes patterns

As stated in 1st law of software evolution by Lehman and Belady [59], a system has to undergo continuous

change in order to remain satisfactory for its stakeholders. Source code is one of the important artifacts that can be accessed from source code repository at any stage (i.e. version) in the history of the software evolution. Source code version histories contain wealth of information that how source code evolve during development. Information regarding changes made to fix a problem, accommodating new changes, adding new feature are recorded. When a programmer is changing a piece of code, they want to determine which related files or routines are updated to be consistent with these changes. To help identifying the relevant parts of the code for a given task, there is need of tools and techniques that statically or dynamically analyzes dependencies between parts of the source (e.g. [60-62]).

Approaches in [26-35], applies data mining techniques on source code control change histories, such as CVS to identify and predict software change. These studies shows that suggestions based on historical co-changes are useful to correctly propose the entities which must co-change.

Zimmermann et al.[28, 32], developed a tool called *Rose* which can guide programmers to locate possible changes by mining historical changes such as source code and related files. The proposed tool uses the association rules extraction technique to identify co-occurring changes by exposing relationship between the modifications of software entities. It aim to answer the question, when a particular source code entity (e.g. a function *A*) is modified, what other entities are also modified (e.g. the functions with names *B* and *C*)? The proposed tool parses the source code and maps the line numbers to the syntactic or physical level entities. These entities are represented as a triple (*filename, type, id*). The subsequent entity changes in the repository are grouped as a transaction. An association rule mining techniques is then applied to determine rules of the form $B,\ C \Rightarrow A$. This information prevents errors due to incomplete changes and finds couplings undetectable by program analysis.

Ying et al.[26], also proposed a technique that also uses association rule mining on CVS version archives. It identifies the change patterns from the source code change history of a system and predict source code change prediction at file level. Each change pattern consists of sets of the names of source files that have been changed together frequently in the past. To provide a recommendation of files relevant to a particular modification task at hand, the developer needs to provide the name of at least one file f_s to generate a set of recommended files f_R.. The files to recommend are determined by querying the patterns to find those that include the identified starting file(s). The usefulness of recommended files f_R is analyzed in term of predictability and interestingness.

Hassan and Holt [29] proposed a method for tracking changes of entities. A variety of heuristics (developer based, history based, code layout based, file based and process based) are proposed which are used to predict the entities that are candidates for a change on account of a given entity being changed. CVS annotations are lexically analyzed to derive the set of changed entities from the source code repositories.

There is the rich literature regarding bug detection and prediction by mining historical data [36, 37, 63-68]. These studies mine history data to find pattern in bug fix changes. The tool developed by Williams and Hollingsworth [36], DynaMine by Livshits and Zimmermann [63] mines simple rules from software

revision histories. These rules involve mostly method pairs. Williams and Hollingsworth [36, 37] proposed method automatically mine bug-fix information from source code repository to improve bug finding/fixing tools. The type of bug considered was a function-return-value check. It is a two step approach. The first step in the process is to identify the types of bugs that are being fixed many times in source code. The second step is to build a bug detector driven by these findings. The idea is to develop a function return value checker based on the knowledge that a specific type of bug has been fixed many times in the past. Briefly, this checker looks for instances where the return value from a function is used in the source code before being tested. The checker does a data flow analysis on the variable holding the returned value only to the point of determining if the value is used before being tested. It simply identifies the original variable and determines the next use of that variable. Code checker is used to determine when a potential bug has been fixed by a source code change. It runs over both versions of the source code. If for a particular function called in the changed file the number of calls remains the same and the number of warnings produced by tool decreases, the change is said to fix a bug.

Livshits and Zimmermann [63] use source code versions to mine call usage patterns by using itemset mining. A tool DYNAMINE is proposed which determine useful usage patterns e.g. call pairs. They classified the mined patterns into valid patterns, likely error patterns, and unlikely patterns with additional dynamic analysis. A candidate pattern mined from the version archive considered to be a valid pattern if it is executed a specified number of times and an unlikely pattern otherwise. Likewise, if a valid pattern is also violated (i.e., only a proper subset of the calls are executed) a large number of times, it is considered as an error pattern. Their approach is more specific in finding violation patterns on method usage pairs. For example, *blockSignal()* and *unblockSingal()* should always be paired in the source code. In addition to the standard ranking methods they also presented a corrective ranking (i.e. based on past changes that fixed bugs) to order the mined patterns. The approach is validated on Eclipse and jEdit systems. The results indicate that their approach along with the corrective ranking is effective in reporting error patterns.

Williams et al. [64] proposed a method that automatically mine function usage patterns and detect software bugs via static analysis of a single version and evolutionary changes. The patterns specifically considered are the patterns called after (i.e. a function *B* is called after function *A*) and conditionally called after (i.e. a function *B* is called after function *A,* but guarded by a condition). Mining the source code repository identifies the instances of such usage patterns. The goal was to find new instances in the current version. Function calls are identified by using C parser. A function usage pattern is the pair of function call found within a distance specified by the number of lines of code. When a function returns a value, using the value without checking it may be a bug.

Kim et al. [69] built BugMem a project-specific bug finding tool which detects potential bugs by analyzing the history of bug fixes and suggests corresponding fixes. It mines bug fixes from software repositories to reconstruct pairs of bug and fix patterns. To construct patterns of defects and their fixes it checks all kind of component in changed region and suggest correct code to repair detected buggy code.

3.5 Mining source code for other purpose

This section provides an overview of mining approaches used to assist with various SE tasks by using any kind of software engineering data.

An approach is proposed in [70] that exploits association rules extraction techniques to analyze defect data. Software defects include bugs, specification and design changes. The collected defect data under analysis are nominal scale variables such as description of defect, priority to fix a defect and its status as well as interval and ratio scale variable regarding defect correction effort and duration. An extended association rule mining method is applied to extract useful information and reveal rules associated with defect correction effort.

Tjortjis et al. [71] employ association rule mining on source code for grouping together similar entities within a software system. The item set used by them consists of variables, data types and calls to blocks of code (modules), where modules may be functions, procedures or classes. The transaction set thus consists of variables, types accessed and calls made by modules. In their algorithm large item sets are first generated by finding item sets that have a higher support than a user-defined threshold. From this item set association rules with confidence greater than a user-defined threshold are generated. Finally groups of modules are created based on the number of common association rules. A variety of techniques are proposed by applying data mining on source code entities for program comprehension and architecture recovery to support software maintenance [72-80].

Kanellopoulos et al.[73] Proposed a framework for knowledge acquisition from source code in order to comprehend an object oriented system and evaluate its maintainability. Clustering techniques are used to understand the structure of source code and assessing its maintainability. The proposed framework works by extracting entities and attribute from source code and constructs input model. Another part of the framework is an extraction process which aim to extract elements and metrics from source code. Extracted information is stored in a relational database to apply data mining techniques. Clustering techniques are applied to analyze the input data and provide a rough grasp of the software system for maintenance engineer. Clustering produces overviews of systems by creating mutually exclusive groups of classes, member data and methods based on their similarities.

Pinzger and Gall uses code patterns to recover software architecture [72]. In their approach user specify the code patterns (text and structural information of source code) by describing their association. Based on specified pattern definition input patterns are matched with source files to reconstruct higher-level patterns that describe the software architecture.

Mancoridis et al.[80] Proposed an automatic technique that creates a hierarchical view of the system organization based solely on the components and relationships that exist in the source code. The technique extracts the Modular Dependency Graph (MDG) from source code in order to identify significant connection among the system modules and stores in database. A textual representation of MDG is obtained

by querying the database. Clustering is performed on MDG that aims to partition the components of a system into compact and well separated clusters.

Sartipi et al. [81] uses both association rule mining and clustering to identify structurally related fragments in the architecture recovery process of legacy system. The source code of a legacy system is analyzed and a set of frequent itemsets is extracted by using clustering and pattern matching techniques. The proposed algorithm defined the components of the legacy system and the best matching component of the system is selected upon user query. Also scores are associated with each possible answer (match) to the user query and thus a ranking of design alternatives can be presented to the user for further evaluation.

4. Analysis of surveyed tools and techniques

The approaches surveyed in section 3 have a number of common characteristics. They all are working on source code data at some level of software granularity e.g. functions, procedures, classes, variables, data types, structure and files. All extract pertinent information from source code analyzes this information and derive conclusions within the context of particular SE tasks. We have organized the surveyed approaches in term of four main facets: mining approach, input, results and SE task benefited. Table 1 organized the survey approaches on following four common facets.

- *Mining approach* entails the algorithm used by proposed technique. Different algorithm used in source code mining research from data mining domain. For example, Frequent item-set mining [2, 82, 83] which find frequent item-set in large database, frequent subsequence mining [4] which find all the frequent subsequences from sequential database Suffix tree based matching and graph matching algorithms.
- *Input* criterion shows which elements of source code are used as input by data mining tool such as functions, classes, variables, data types etc.
- The criterion *Results* reflects which type of mining information are extracted by each approach e.g. programming rules, copy paste code, reuse component.
- We also included the specific *SE task* that each approach addressed. This gives a general context to researchers who are interested in specific task being benefited from these approaches.

Table 1

Analysis of surveyed approaches

One sentence description	Mining approach	input	Result	SE task	Author/Ref.
Mining rules to detect bugs					
Checking rules against program code and cross check for contradiction	Statistical analysis	Functions	Pair-wise programming rules.	Programming, defect detection	Engler at el. [7]
Extract implicit programming rules and detect their violations.	Item-set mining	Functions, variable, data types	Pair-wise and complex rules	Programming, defect detection	Li, and Zhou [8]
Infer function precedence protocols and their violation	Frequent sub-sequence mining	Functions	Function calls ordering rules	Programming, defect detection	Ramanathan et al.[9]
Identify conditional rules and their violations	Frequent item-set sub-graph mining	PDG	Graph minor as conditional rules	Programming, defect detection	Chang et al.[10]
Extract variable correlations rules	Frequent item-set mining	Functions, variables	Variable pairing rules	Programming, defect detection	Lu et al.[11]
Itemset vs. sequential pattern mining to the number of patterns, and violations	Frequent pattern, association rule	Function definition.	Call usage patterns	Programming, defect detection	Kagdi et al.[44]
Mining code clones patterns					
Suffix trees for tokens per line	Suffix tree based matching	Sequence of lines	Line by line clones	Maintenance	B. Baker [12]
Token normalizations, then suffix-tree based search	Token comparison, tree matching	Sequence of tokens	Clone pairs	Maintenance	Kamiya et al.[13]
Data mining for frequent token sequences	Frequent subsequence and tokenization	Statement sequence	Copy-paste code fragment	Programming, defect detection	Li et al.[53]
Searching clones in general tree structures	Frequent item set mining	XML of ASTs	Clone pairs	Maintenance, refactoring	Wahler et al. [15]
Searching similar sub graphs in PDGs	Spatial search, graph matching algorithm	PDG	Matching sub-graph	Maintenance defect detection	Qu et al.[17]
Mining for frequently applied changes in a version control system.	Parameterized token matching	Source code changes from CVS deltas	Similar pairs of source code changes	Development, maintenance	Rysselberghe and Demeyer [55]
Structural clone detection at different level of abstraction	Frequent item-set, clustering	Source code files	Structure clone	Maintenance, refactoring	Basit and Jarzabek [54]
Mining API Usage patterns					
Application classes inheriting from a particular library class often instantiate another class or one of its descendants.	Item-set and association-rule mining	Components, classes, and functions	Library reuse pattern	Code reuse	A. Michail [58]
Locates a set of relevant code examples from an example repository.	Heuristic matching	Structural context of code	List of relevant code example under development	Code reuse, programming	Holmes and Murphy [20]

Synthesize API code from source method to destination	Signature graph matching	API Method signature/class type	Synthesize API code	Code reuse, programming	Mandelin et al.[23]
Mining sample code repository for relevant code	Graph mining	Inheritance hierarchy, fields and methods	API code snippets	Code reuse	Sahavechaphan et al.[22]
Mines segments returned by a code search engine	Frequent sequence mining	Method, class or package	Sequencing information among method calls	Code reuse	Xie, and Pei [25]
Search web dynamically for related code and mine the return code to build MISs	Clustering	Objects	MIS and relevant code sample	Code reuse, programming	Thummalapenta, and Xie [24]
Mining co – change and bug fixes change pattern					
Searches for a commonly fixed bug	Static analysis, text retrieval	Function return value	Functions involved in a potential bugs	Programming, bug detection	Williams, and Hollingsworth [36, 37]
Call usage patterns and their violation	Itemset mining	Methods	Call usage patterns	Programming, bug detection	Livshits, and Zimmermann [63]
Detect function usage patterns and detect software as violation of usage patterns	Static analysis	Function calls	Function usage pattern	Programming, bug detection	Williams et al. [64]
Mines version archives to make changes recommendations for programmers.	Frequent pattern mining and association rules	Files, function, variables	Prediction of failures, Correlations between entities	Development, maintenance	Zimmermann et al.[28, 32]
Extracts source files that tend to change together	Frequent pattern and correlation set mining	Files	Prediction of the set of co- changed files	Development maintenance	Ying et al.[26]
Propose several heuristics to identify how change propagate	Mining via heuristic	Function, variable, or data type	Predicting candidate entities for change	Development maintenance	Hassan and Holt [29]
Detecting project specific bugs by analyzing the history of bug fixes	Static analysis	Project change history	Warning message and suggest changes	Bug detection, development	Kim et al. [69]
Mining code for other purpose					
Defect analysis to reveals rules associated with defect correction efforts	Frequent pattern and association rules	Defect data	Defect correction effort rules	Defect correction	Morisaki [70]
Grouping together similar entities within a software system.	association rules	Functions, procedures, classes	Groups of similar modules	Maintenance	Tjortjis et al.[71]
Program comprehension by recovering structure of source code	Clustering	Classes, method and attributes	Groups of similar classes, methods, data	Maintenance	Kanellopoulos at al.[73]
Recovering software architecture by extracting code patterns	String matching	Variables, functions and structures	Higher-level views of the software system	Architecture recovery	Pinzger and Gall [72]
Partition the software system	Clustering	MDGs	Modular structure	Development, maintenance	Mancoridis et al. [80]
Identify structurally related fragments in the	Frequent pattern, association rules	Legacy system	Design alternatives	Architecture recovery	Sartipi et al.[81]

architecture process	recovery

5. Data mining techniques used to mine source code

This section organizes the surveyed approaches in term of data mining technique used. To apply mining algorithm source code is first transformed into format suitable for particular mining algorithm. This is done by extracting relevant data from the raw source code data for example, static method call sequences or call graphs. This data is further processed by cleaning and properly formatting it for the mining algorithm. For example, the input format for transaction data can be a transaction database where each transaction contains sets of items. In general, mining algorithms fall into following main categories:

- *Frequent pattern mining.* Finding frequently occurring patterns.
- *Association rules mining.* Find association among frequently occurring patterns
- *Sequential pattern mining.* Finding commonly occurring sequential patterns.
- *Pattern Matching.* Finding data instances for given patterns.
- *Clustering.* Grouping data into clusters
- *Graph Mining.* Graph mining algorithms includes: frequent sub-graph mining, graph matching, graph classification, and graph clustering.
- *Classification.* Predicting labels of data based on the already labeled data.

Table 2 organizes the surveyed approaches in term of data mining techniques they used.

Table 2

Organization of surveyed approaches by data mining techniques

Mining Technique	Approaches
Itemset mining	Li, and Y. Zhou [8], Chang et al.[10], Lu et al.[11], Li et al.[53], Wahler et al. [15], Zimmermann et al.[28, 32], Kagdi et al. [44], Ying et al.[26], Livshits, and Zimmermann [63], Basit and Jarzabek [54], Sartipi et al.[81], Morisaki [70], Kagdi et al.[44], Basit and Jarzabek [54], A. Michail [58]
Association rules	A. Michail [58], Zimmermann et al.[28, 32], Sartipi et al.[81], Tjortjis et al.[71], Morisaki [70], Song et al.[66], Kagdi et al.[44]
Sequential pattern mining	Kagdi et al. [44], Xie, and Pei [25], Li et al.[53]
Clustering	Basit and Jarzabek [54], Kanellopoulos at al.[73], Mancoridis et al. [80], Basit and Jarzabek [54], Thummalapenta, and Xie [24]
Graph mining	Chang et al.[10], Sahavechaphan et al.[22], Mandelin et al.[23], Qu et al. [17]
Pattern matching	Pinzger and Gall [72], B. Baker [12], Mandelin et al.[23], Kamiya et al.[13], Rysselberghe and Demeyer [55]
Static analysis	Williams, and Hollingsworth [36, 37], Engler at el. [7]

Mining via heuristic	Holmes and Murphy [20], Hassan and Holt [29]

Note: The data in table 1, 2 and 3 is based on material published in papers

6. Comparison of source code mining tools

An overall analysis of the tools and techniques with respect to several general characteristics is shown in Table 1. This section compares those approaches which developed a supporting tool 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. Such a tool can predict and suggest probable changes to the source code.

Tool Availability indicates whether there is documented IDE support for the method/tool. Only a few methods provide direct IDE support. Most of the tools are not freely available hence hampers the quantitative analysis of tools.

The *External Dependencies* indicates whether the tool requires other language, environment or tools to work, for example PR-Miner[8] and MUVI require parser to convert source code into item-set database. CCFinder requires language-dependent transformation rules and likewise other tools also have some external dependency.

Language support indicates the languages supported by tools. We can observe that there are very few tools that are aimed at OO-languages (e.g., C++).

Algorithm/ technique, Identifies the different algorithms used in source code mining research from other domains. For example, the suffix-tree algorithm computes all of the same subsequences in a sequence composed of a fixed alphabet (e.g. characters, tokens and hash values of lines) in linear time and space. It can only handle exact sequences. On the other hand data mining algorithms are well suited to handle arbitrary gaps in the subsequences. Apart from data mining techniques other analysis techniques are also observed such as Engler et al. work uses two techniques *Internal Consistency* which finds errors where programmers have violated beliefs that must hold and *Statistical Analysis* extracts beliefs from a much noisier sample where the extracted beliefs can be either valid or coincidental.

The *Empirical Validation* criterion shows the kind of validation that has been reported for each tool and *Availability of Empirical Results* indicates whether the results of the validations are available. The last criterion, *Evaluation*, indicates the common systems that have been used as experiment to run the tool. Table 3 compares the surveyed approach in term of tool developed.

Table 3

Comparison of surveyed approach in term of tool developed

Note: *Avail = Availability, Ext. Depend. = External dependency, Lang. = Language support, OO = Object Oriented, P= Procedural and Y= Yes, N = No*

	Tool	**Avail.**		**Ext. Depend.**		**Lang.**		**Algorithm**								**Empirical validation**			**Result Availability**			**Evaluation**			
		Y	N	Y	N	OO	P	Apriori	FPClose	CloSpan	FP- growth	Clustering	Pattern matching	Graph mining	Statically analysis	Moderate	Partial	Validate well	Full	Partial	Not conclusive	Preliminary	Casa study	Industrial	Other
Engler at el. [7]	Checker		◦	◦			◦								◦	◦				◦				◦	
Li, and Zhou [8]	PR - Miner			◦			◦	◦	◦									◦		◦				◦	
Ramanathan et al.[9]	Chronicler		◦		◦		◦	◦								◦				◦				◦	
Lu et al.[11]	MUVI		◦	◦		◦	◦		◦									◦		◦				◦	
B. Baker [12]	Dup		◦		◦	◦							◦				◦			◦		◦			
Kamiya et al.[13]	CCFinder	◦			◦		◦						◦					◦	◦					◦	
Li et al.[53]	CP Miner		◦	◦		◦	◦			◦								◦		◦				◦	
Wahler et al. [15]	CloneDetection		◦		◦	◦		◦								◦				◦				◦	
Basit, and Jarzabek [54]	Clone Miner		◦		◦						◦	◦						◦	◦				◦		
A. Michail [58]	CodeWeb		◦		◦	◦		◦								◦				◦			◦		
Holmes and Murphy [20]	Strathcona		◦	◦		◦	◦										◦			◦		◦			
Mandelin et al.[23]	Prospector	◦		◦		◦								◦				◦		◦		◦			
Sahavechaphan et al.[22]	XSnippet		◦		◦	◦										◦				◦					◦
Xie, and Pei [25]	MAPO	◦			◦	◦		◦								◦					◦				◦
Thummalapenta, and Xie [24]	ParseWeb	◦			◦	◦						◦						◦		◦					◦
Williams at al. [36]	Static Checker		◦		◦	◦									◦			◦		◦			◦		
Livshits and Zimmermann [63]	DYNAMINE	◦			◦	◦		◦										◦		◦				◦	
Kim et al. [69]	BugMem		◦	◦		◦									◦		◦			◦				◦	

7. Critique on source code mining approaches

Table 1 shows most of approaches used frequent pattern mining techniques to identify patterns from source code. By making comparison between these techniques we can identify strength and limitations of these techniques.

7.1 Mining rules to detect bugs

Engler et al. [7] approach relies on developers to supply rule templates such as function *A* must be paired with function *B* and corresponding checkers. Since such template-based methods only cover the given or explicit rules known in advance, it may miss many violations due to the existence of implicit rules. Moreover, It only performs one type of pattern analysis such as: "function *A* should be paired with function *B*" and does not consider other semantic dependencies.

PR-Miner [8] 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. However, PR-Miner does not consider relevant constraints between rule elements and so apparently will identify a set of elements that frequently appear together in functions as a possible rule without other evidence that the elements are semantically related. 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. Also numbers of false positives are increased as no inter-procedural analysis is used. Both Engler *et al.* work and PR-Miner discover patterns involving 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.

These limitations are addressed by inter-procedural path-sensitive static analysis tool CHRONICLER [9] which is fundamentally different from PR-Miner as it ensures path-sensitivity hence generate less number of false negative as compared to PR-Miner. Since CHRONICLER computes association of specific function rather than entire program hence reduces the number of protocol generated by eliminating false positive as reported by PR- Miner. It differ from Engler et al. [7] approach as it computes the precedence relationship based on program's control flow structure whereas, Engler et al. work detects relations between pairs of functions by exploiting all possible paths. However, CHRONICLER does not take data flow or data dependency into account. A new approach to discovering implicit conditional rules [10]

addresses this limitation by transforming program units into program dependency graph which captures data and control flow dependencies as well as other essential constraint among program elements. 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, multi-graphs (multiple edge between given pair of node) and require the modification of graphs. Modifications such as ignoring edge directions or replacing a call site graph with a single node may cause information loss so that precision is sacrificed in rule discovery. Moreover the approach considered only a small set of nodes in PDGs, and the patterns are only control points in a program.

All of the approaches mentioned above focused on procedures and component interfaces instead of variable correlations where as MUVI [11] mines variable correlations and generate variable-pairing rules. Engler et al. [7] also detect variable inconsistency through logical reasoning for example, some statement indicates that a pointer might be NULL but a subsequent statement assumes that pointer must not be NULL so a conflict arise where as MUVI [11] detect inconsistencies using pattern analysis on multi-variable access correlations.

7.2 Mining code clones patterns

Dup[12] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed syntactically identical clones whereas, CCFinder [13] 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 computational complexity since a single line is usually composed of several tokens. Token based methods have intrinsic limitation for pattern mining of cloned codes due to using only spatial space analysis such as reordered or inserted statements can break a token sequence which may otherwise be regarded as a duplicate to another sequence. CloneDetection search clones in general tree structure and works on abstract level as compared to others. 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 [53] applies data mining technique to identify similar sequence of tokenized statements rather than token comparison and 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 [17] 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. However, graph-based techniques are limited in scalability. All the mentioned clone detection techniques detects simple clones i.e. fragment of duplicated code and not looking at the big picture where these fragments of duplicated code are possibly part of a bigger replicated program structure. In contrast *Clone Miner* [54] performs further analysis on simple clones that co-exists and relates to each other in certain way.

Simple clone detectors usually detect clones larger than a certain threshold (e.g., clones longer than 5 LOC). Higher thresholds risk false negatives, while lower thresholds detect too many false positives. In comparison, *Clone Miner* can afford to have a lower threshold for simple clones, than a stand-alone simple clone detector, without returning too many false positives. This is because it can use the grouping as a secondary filter criterion to filter out small clones that do not contribute to structural clones. However, like other approaches it also detects clone based on physical location of clones and not detect semantic associations among clones.

7.3 Mining API usage patterns

CodeWeb demonstrate how the library classes have been reused in existing applications. To get this information, a developer must populate CodeWeb with applications that are similar to the one which they are developing. To use CodeWeb developer must find similar applications of interest in advance. It also uses the structural attributes to compare complete projects against one another instead of enabling the use of fragments of projects. The need to find applications in advance suggests that a developer would be more likely to engage in the use of CodeWeb at the beginning of the development process as it is based on browsing rather than querying.

Given an API sample, Strathcona, Prospector, XSnippet , MAPO and Parseweb provide example code of that API. Strathcona suggest similar code examples stored in an example repository by matching the context of the code under development with the samples stored in the example repository. Strathcona generates relevant solutions when the exact API is included in the search context but mostly programmer has no knowledge of which API has to be used for solving the query. It is based on heuristics which are generic and generate many irrelevant examples.

Prospector tries to solve the queries related to a specific set of frameworks or libraries by using API signatures. As API signatures are used for addressing the query, Prospector returns many irrelevant examples. Strathcona, for example, does not specialize the heuristics it employs based on the developer's context and its results straddle the extremities – in some cases providing too many irrelevant results while in others over-constraining the context to provide too few or no results. Prospector, while performing better than Strathcona in general has its own limitations. First, its over-reliance on API information can result in too many irrelevant results. For example, two unrelated paths can be connected by generic classes such as Object and *ArrayList* discounting the diversity in their semantics. Second, the context description is limited to only visible input types of fields declared in the boundary of method and class while context information such as the "parent" is ignored thereby missing a set of potentially qualified hits. Prospector can generate compilable code for its suggested solutions.

XSnippet simply returns the set of all code samples contained in the sample repository that instantiate the given *destination object* type, irrespective of the *source object* type. Moreover, XSnippet is also limited to the queries of a specific set of frameworks or libraries. Strathcona and XSnippet use the code relevance to

define the code context which best fit the required code. However Strathcona only use the lexically visible types to define the code relevance where as XSnippet uses parents of the class under development as well as lexically visible types for a given method contained in class to define code relevance. The major problem with both of approaches is the availability of limited code samples stored in the repository.

MAPO defines a query that describes a method, class, or package for an API, the tool can gather relevant code samples from open source repositories and conduct data mining. It can extract common patterns among the list of relevant code examples returned by a code search engine. It does not synthesized code that can be directly inserted into developers' code. For using MAPO Programmers need to know the API to be used to identify usage patterns of that API.

PARSEWeb like MAPO takes queries of the form "source object type to destination object type" as an input and suggests what API method sequence should be used to obtain one object from another potential solution. PARSEWeb search web dynamically for relevant solution and not limited to the queries of any specific set of frameworks or libraries like Prospector and XSnippet . Parseweb uses code sample for solving given query hence identifying more relevant code sample. Prospector which solves the queries through API signatures and has no knowledge of which MISs is often used compared to other MISs that can also serve as a solution for the given query. PARSEWeb performs better in this scenario because it tries to suggest solutions from reusable code samples and is able to identify MISs that are often used for solving a given query. However, PARSEWeb suggests only the frequent MISs and code samples, but cannot directly generate compilable code. Neither PARSEWeb nor Prospector considers the code context.

7.4 Mining co-changes and bug fix changes patterns

Zimmermann et al.,[28] and Ying et al., [26] both uses association rule mining on CVS data to mine co-change patterns i.e. is potentially relevant piece of code to a given fragment of source code. However, in Zimmermann et al., approach resultant rules must satisfy some support and confidence. In this way it can give misleading association rules in cases where some files have changed significantly more often than others. Whereas, Ying et al. only uses the support value and additional correlation rule mining, which takes into account how often both files are changing together as well as separately. Both approaches produce similar quantitative results. The qualitative analyses differ. Zimmermann et al. present some change associations that were generated from their approach and argue that these associations are of interest. In contrast, Ying et al. especially evaluated the usefulness of the results by analyzing the recommendations provided in the context of completed modification tasks. Moreover, Zimmermann et al., approach suggests the fine grained entities (method and classes) that changed together provide better results because smaller units of source code suggest a similar intention behind the separated code. In contrast Ying et al. approach change patterns describes files that change together repeatedly. In comparison Hassan and Holt [29] proposed tracking changes of more fine grained entities, namely, function, variable, or data type, to determine how changes propagate from one entity to another.

The work by Williams, and Hollingsworth [36, 37, 64] and Dynamine [63] combined revision history mining and program analysis to discover common error patterns. Williams, and Hollingsworth [36, 37] claims if function return value without first checking its validity is used it may lead to a latent bug. In practice, this approach leads to many false positives, as typical code has many locations where return values are used without checks. Moreover, they focus on prioritizing or improving only existing error patterns. Instead Dynamine [63] concentrate on discovering new patterns and dynamic analysis of detected patterns leads to less number of false positive. Additionally, Williams, and Hollingsworth [64] also mines version histories for detecting bugs in method usage pair by focusing only on pair of function used together and their violation, in contrast Dynamine uses usage patterns of functions to detect violation.

All the previous bug detection techniques searches for predefined common bug patterns such call usage or method pair. In contrast BugMem [69] learned from previous bug fix change in specific project so the bug patterns are project specific, and project-specific bugs can be detected. However, it only considered bug fixing patterns and does not consider the changes which may introduce new bugs.

8. Discussion and future trends

A wide range of research has been done in the area of checking and enforcing specific coding rules, the violation of which leads to well-known types of bugs. Since data mining algorithms are traditionally meant for large dataset stored in database or warehouse and not directly applicable on source code. A great deal of time and effort has been spent by researcher to find worthwhile rules due to the complexity of data extraction and preprocessing methods. Most of bug detection techniques are application specific. As a result lesser known types of bugs and applications remain virtually unexplored in error detection research. A better approach is needed if we want to test new or unfamiliar applications with error detection tools. Furthermore, the rule mining approaches detects general programming rules from source code. The performance could be improved if domain specific information is combined and some knowledge about specific rules is provided to rule mining technique to extract only of them from source code. It could notably increase the accuracy and efficiency. Moreover, bug finding techniques solely relies on source code data or historical changes. In this way bug reported by the tester become isolated from development team. Based on our observation a system of bug finding techniques which correlates bug reports and the corresponding source code changes might be helpful.

There is lot of studies related to find bug fixes from history data. All are focus on one aspect of change e.g. bug fixes change [69]. All types of change pattern could provide useful information to the developers when they are changing their code. The developers need to use multiple techniques on one project to find impact of change. However finding all existing program file change patterns like bug fixing, bug introducing and bug fix introducing might be helpful. We concludes there is a strong need of light weight approach to use prior bug finding techniques together to maximize bug detection capability.

Also a variety of architecture recovery techniques are available. A common idea is to integrate several tools in architecture workbenches. In this way a variety of techniques will available in one umbrella to examine a system and extract static and dynamic views to reconstruct a system.

We have analyzed the major applications of clone detection. It signals weak points in the program and encourages the *restructuring* and *refactoring*. A fully automatic replacement of clones by higher order structures is certainly not the best choice. But in this aspect integration with an interactive program development environment would be very helpful.

9. Conclusion

In this paper we have provided concise but comprehensive survey of state of art source code mining tools and techniques. So far this is first survey which includes combination of different techniques. Comparison of techniques and tools shows, there is a no single technique which is superior to all other in every aspects because all techniques have strength and weaknesses and intended for different task and context. The comparison also helps how to employ a set of different tools to achieve better results.

The results of this survey show all the previous studies mines a specific pattern types to accomplish a certain SE task. Thus, programmers need to employ multiple methods to mine different kind of useful information which increase computational cost and time. However, SE tasks increasingly demand the mining of multiple correlated patterns together to achieve the most effective result. Based on our observation a hybrid light weight tool is required. The tool should extract multiple patterns from source code and applies data mining techniques at different layers to assists in multiple software engineering tasks over different phases of development life cycle e.g. assisting programming in writing code, bug detection and software maintenance. In this direction we are working on development of light weight tool to extracts a variety of patterns from source code as presented in[84]. The work is in its initial stage and in future further research would enrich in context of pattern findings and their violation.

References

[1] J. Shafer, R. Agrawal, and M. Mehta, “SPRINT: A scalable parallel classifier for data mining,” in *Proceedings of the twenty-second international conference on very large databases*, 1996, pp. 544-555.

[2] R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *Proceedings of the 20th International Conference on Very Large Data Bases.Morgan Kaufmann Publishers Inc*, 1994, pp. 487-499.

[3] R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules,” in *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, 1994, pp. 487-499.

[4] R. Agrawal, and R. Srikant, “Mining sequential patterns,” in *Eleventh International Conference on Data Engineering*, 1995, pp. 3.

[5] A. E. Hassan, “The road ahead for mining software repositories,” in *Proceedings of the frontiers of software maintenance (FoSM' 08)*, 2008, pp. 48-57.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] B. Baker, "On finding duplication and near-duplication in large software systems," in *Second IEEE Working Conf on Reverse Eng.(wcre)*, 1995, pp. 86-95.

[13] 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.

[14] I. Baxter, A. Yahin, L. Moura *et al.*, "Anna and L. Bier,"Clone Detection Using Abstract Syntax Trees," Proc. Int" l Conf," *Software Maintenance*, pp. 368-377, 1998.

[15] 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.

[16] J. Krinke, "Identifying similar code with program dependence graphs," in Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, 2001, pp. 301-309.

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

[18] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Communications of the ACM,* vol. 39, no. 10, pp. 116, 1996.

[19] G. T. Heineman, and W. T. Councill, *Component-based software engineering: putting the pieces together*: Addison-Wesley USA, 2001.

[20] 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.

[21] G. T. Leavens, and M. Sitaraman, *Foundations of component-based systems*: Cambridge Univ Press, 2000.

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

[23] 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.

[24] 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.

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

[26] A. T. T. Ying, G. C. Murphy, R. Ng *et al.*, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering,* vol. 30, no. 9, pp. 574-586, 2004.

[27] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, "Supporting maintenance of legacy software with data mining techniques," in *Proc.Conf. the Centre for Advanced Studies on Collaborative Research*, 2000, pp. 11.

[28] T. Zimmermann, P. Weisgerber, S. Diehl *et al.*, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering,31(6)*, pp. 429-445, June 2005.

[29] A. E. Hassan, and R. C. Holt, "Predicting change propagation in software systems," in *Proceedings of the 20th IEEE International Conference onSoftware Maintenance*, 2004, pp. 284-293.

[30] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proc. Working Conf. Reverse Eng.*, 2003, pp. 90-99.

[31] A. Mockus, and D. M. Weiss, "Globalization by chunking: a quantitative approach," *Software, IEEE,* vol. 18, no. 2, pp. 30-37, 2001.

[32] T. Zimmermann, P. Weisgerber, S. Diehl *et al.*, "Mining version histories to guide software changes," in *Proceedings of the 26th international conference on Software Engineering (ICSE'04)*, 2004, pp. 563-572.

[33] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of International Conference on Software Maintenance*, 1998, pp. 190-198.

[34] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *Proceedings on Sixth International Workshop on Principles of Software Evolution.*, 2003, pp. 13-23.

[35] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in OO software through visualization," in *Proceedings of the 11th Internationa Workshop on Program Comprehension*, 2003, pp. 44-53.

[36] C. C. Williams, and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, pp. 466-480, 2005.

[37] C. C. Williams, and J. K. Hollingsworth, "Bug driven bug finders," in *Int'l Workshop Mining Software Repositories (MSR '04)*, May 2004, pp. 70–74.

[38] S. Khatoon, A. Mahmood, and G. Li, "An evaluation of source code mining techniques," in *Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)* 2011, pp. 1929-1933.

[39] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 19, no. 2, pp. 77-131, 2007.

[40] M. Halkidi, D. Spinellis, G. Tsatsaronis *et al.*, "Data mining in software engineering," *Intelligent Data Analysis,* vol. 15, no. 3, pp. 413-441.

[41] T. Ball, and S. K. Rajamani, "The S LAM project: debugging system software via static analysis," *ACM SIGPLAN Notices,* vol. 37, no. 1, pp. 1-3, 2002.

[42] D. L. Heine, and M. S. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '03)*, June 2003, pp. 168-181.

[43] D. Burdick, M. Calimlim, J. Flannick *et al.*, "Mafia: A performance study of mining maximal frequent itemsets," in *Workshop on Frequent Itemset Mining Implementations (FIMI' 03)*, 2003.

[44] H. Kagdi, M. Collard, and J. Maletic, "Comparing Approaches to Mining Source Code for Call-Usage Patterns," in *International Conference on Software Engineering: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[45] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance*, 1996, pp. 244-253.

[46] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1999, pp. 109.

[47] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming,* vol. 74, no. 7, pp. 470-495, 2009.

[48] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 49-49, 1993.

[49] M. Balazinska, E. Merlo, M. Dagenais *et al.*, "Advanced clone-analysis to support object-oriented system refactoring," in *Seventh Working Conference on Reverse Engineering*, 2000, pp. 98.

[50] I. D. Baxter, A. Yahin, L. Moura *et al.*, "*Clone detection using abstract syntax trees*," in ICSM, 1998, pp. 368.

[51] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of Centre for Advanced Studies on Collaborative research: software engineering*, 1993, pp. 171-183.

[52] R. Komondoor, and S. Horwitz, "Using slicing to identify duplication in source code," *Static Analysis*, pp. 40-56, 2001.

[53] 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*, 2004, pp. 20.

[54] H. A. Basit, and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software Engineering*, pp. 497-514, 2009.

[55] F. Van Rysselberghe, and S. Demeyer, "Mining version control systems for FACs (frequently applied changes)," in *Int'l Workshop on Mining Software Repositories (MSR '04)*, May 2004, pp. 48-52.

[56] H. A. Basit, S. J. Puglisi, W. F. Smyth *et al.*, "Efficient token based clone detection with flexible

tokenization," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2007, pp. 513-516.

[57] 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.

[58] 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.

[59] M. M. Lehman, and L. A. Belady, *Program evolution: processes of software change*: Academic Press Professional, Inc., 1985.

[60] H. Agrawal, and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices,* vol. 25, no. 6, pp. 246-256, 1990.

[61] D. W. Binkley, and K. B. Gallagher, "Program slicing," *Advances in Computers,* vol. 43, pp. 1-50, 1996.

[62] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, 1981, pp. 439-449.

[63] B. Livshits, and T. Zimmermann, "DynaMine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes,* vol. 30, no. 5, pp. 296-305, 2005.

[64] C. C. Williams, and J. K. Hollingsworth, "Recovering system specific rules from software repositories," in *International Workshop on Mining Software Repositories (MSR'05)*, St. Louis, 2005, pp. 1-5.

[65] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering,* vol. 14, no. 3, pp. 286-315, 2009.

[66] Q. Song, M. Shepperd, M. Cartwright *et al.*, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, pp. 69-82, 2006.

[67] R. Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Transactions on Software Engineering*, pp. 579-596, 2008.

[68] B. Turhan, G. Kocak, and A. Bener, "Data mining source code for locating software bugs: A case study in telecommunication industry," *Expert Systems with Applications,* vol. 36, no. 6, pp. 9986-9990, 2009.

[69] S. Kim, K. Pan, and E. Whitehead Jr, "Memories of bug fixes," in Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 2006, pp. 35-45.

[70] S. Morisaki, A. Monden, T. Matsumura *et al.*, "Defect data analysis based on extended association rule mining," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007, pp. 3.

[71] C. Tjortjis, L. Sinos, and P. Layzell, "Facilitating program comprehension by mining association rules from source code," in *Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)*, 2003, pp. 125-132.

[72] M. Pinzger, and H. Gall, "Pattern-supported architecture recovery," in *10th International Workshop on Program Comprehension (IWPC'02)*, 2002, pp. 53-61.

[73] Y. Kanellopoulos, T. Dimopulos, C. Tjortjis *et al.*, "Mining source code elements for comprehending object-oriented systems and evaluating their maintainability," *ACM SIGKDD Explorations Newsletter,* vol. 8, no. 1, pp. 33-40, 2006.

[74] C. M. de Oca, and D. L. Carver, "Identification of data cohesive subsystems using data mining techniques," in *Int'l Conf. Software maintenance (ICSM 98)*, 1998, pp. 16.

[75] D. Rousidis, and C. Tjortjis, "Clustering data retrieved from Java source code to support software maintenance: A case study," *CSMR 05*, 2005.

[76] R. Fiutem, P. Tonella, G. Anteniol *et al.*, "A cliche-based environment to support architectural reverse engineering," in *Proceedings of the Third Working Conference on Reverse Engineering*, 1996, pp. 277-286.

[77] D. R. Harris, H. B. Reubenstein, and A. S. Yeh, "Reverse engineering to the architectural level," in *Proceedings of the 17th international conference on Software engineering*, 1995, pp. 186-195.

[78] K. Sartipi, K. Kontogiannis, and F. Mavaddat, "A pattern matching framework for software architecture recovery and restructuring," in *8th International Workshop on Program Comprehension (IWPC)*, 2000, pp. 37-47.

[79] G. Y. Guo, J. M. Atlee, and R. Kazman, "A software architecture reconstruction method," in *First Working IFIP Conference on Software Architecture (WICSA1)*, *San Antonio, Texas, USA*, 22-24

February 1999, pp. 15.

[80] S. Mancoridis, B. S. Mitchell, C. Rorres *et al.*, "*Using automatic clustering to produce high-level system organizations of source code*," in 6th International Workshop on Program Comprehension, IWPC'98., 1998, pp. 45-52.

[81] K. Sartipi, K. Kontogiannis, and F. Mavaddat, "Architectural design recovery using data mining techniques," in *Proc. European Conf. Software Maintenance and Reengineering*, 2000, pp. 129-139.

[82] G. Grahne, and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," in *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[83] H. Mannila, H. Toivonen, and A. I. Verkamo, "Efficient algorithms for discovering association rules," in *Proceedings of the AAAI workshop on Knowledge Discovery in Databases*, 1994, pp. 181-192.

[84] S. Khatoon, G. Li, and R. M. Ashfaq, "A Framework for Automatically Mining Source Code," *Journal of Software Engineering,* vol. *5 Number 2*, pp. 64-77, 2011.