Journal of Software Engineering, 2011
ISSN 1819-4311 / DOI: 10.3923/jse.2011.

A Framework for Automatically Mining Source Code

[1]Shaheen Khatoon, [1]Guohui Li and [2]Rana Muhammad Ashfaq
[1]School of Computer and Applied Technology, Huazhong University of Science and Technology (HUST), Wuhan, China
[2]Department of Computer Science, International Islamic University, Islamabad, Pakistan

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

ABSTRACT

Mining source code by using different data mining techniques to extract the informative patterns like programming rules, variable correlation, code clones and frequent API usage is an active area of research. However, no practical framework for integrating these tasks has been attempted. To achieve this objective an integrated framework is designed that can detect different types of bugs to achieve software quality and assist developer in reusing API libraries for rapid software development. Proposed framework automatically extracts large variety of programming patterns and finds the locations where the extracted patterns are violated. Violated patterns are reported as programming rule violation, copy paste code related bugs and inconsistent variable update bugs. Although, the bugs are different but the framework can detect these bugs in one pass and produces higher quality software systems within budget. The framework also helps in code reusing by suggesting the programmer how to write API code to facilitate rapid software development. Proposed framework is validated by developing a prototype that developed in C# (MS Visual Studio, 2008) and evaluated on large application like ERP. Results shows proposed technique greatly reduced time and cost of manually checking defects from source code by programmers.

Key words: Source code mining, programming rule, copy-paste code, API usage, constraint-based mining

INTRODUCTION

The primary goal of software development is to deliver high quality software systems within budget and in the least amount of time whenever possible. An efficient way of software testing and creation of new software by reusing existence code is one of the solutions to achieve these goals. Software testing is the critical element of any kind of quality assurance and most important for credibility of software system. Several automatic tools and techniques for software testing are proposed which can automatically detect defects in order to produce high quality software (Lu and Ye, 2007; Maamri and Sahnoun, 2007; Shu *et al*., 2009; Alsmadi, 2011). There are also several studies that discuss techniques to improve test case generation with the goal of improving test coverage to deliver high quality software (Kosindrdecha and Daengdej, 2010; Roongruangsuwan and Daengdej, 2010). All the tools and techniques are relies on requirement or design specification

need to verify against some information. Due to the size and complexity of data repositories now-a-days, Verification Driven approaches are not sufficient to efficiently explore the data available in an organization.

Techniques from other domains such as Artificial Intelligence (Pedrycz and Peters, 1997; Vinayagasundaram and Srivatsa, 2007), Neural network (Khoshgoftaar *et al.*, 1995; Lu and Ye, 2007) and data mining (Hassan and Xie, 2010) can be applied on software engineering data to discover important information hidden in the data. A number of software development issues can be benefited from these disciplines such as: Estimating Software development efforts, software quality and process improvement, software cost estimation etc.

Practitioners are increasingly applying data mining techniques on various software engineering data to improve software productivity and quality. Plenty of studies have been conducted to extract pertinent information and/or uncover relationships from programs source code to support various aspects of software development life cycle such as programming, defect detection, maintenance, management, reusability etc. Most of the studies used static source code analysis to find set of rules or guidelines describing situations of trends or relationships. For example, if A occurs then B and C happen X amount of the time. Program usually follows this kind of implicit programming rules which are not documented and when these rules are violated defects are easily introduced. Therefore, automatic tools are desirable to extracts such kind of rules. Plenty of studies previously done in this direction, for example (Engler *et al.*, 2001) approach and PR-Miner (Li and Zhou, 2005) mine function-pairing rules, CHRONICLER (Ramanathan *et al.*, 2007) mine function precedence protocols and (Chang *et al.*, 2007) mine conditional rules. Some rules indicate variable correlations, i.e., these variables should be accessed together or modified in a consistent manner. Violating access correlations can lead to inconsistent update bugs, that is sometimes, the programmer updates one variable but forgets to update or check its correlated variable. As a result, the memory states of the correlated variables become inconsistent. Therefore, automatic tools are needed to infer variable correlations (Lu *et al.*, 2007).

The significance of data mining on source code is further magnified by detecting copy-paste/clone code. One may be interested to mine code clone/copy-paste code segments to support code optimization and/or to find copy paste code related bugs. 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, such as CCFinder (Kamiya *et al.*, 2002) and Dup (Baker, 1995) that use tokenization on the source code. Abstract syntax tree (Baxter *et al.*, 1998; Wahler *et al.*, 2004) and PGDs (Krinke, 2001; Qu *et al.*, 2010) based tools looks for sub trees and isomorphic graphs to find clones. In addition to above and many other technique we find only two approaches that, CP-Miner (Li *et al.*, 2004) and Clonedetection (Wahler *et al.*, 2004) which uses data mining to detect clones. CP-Miner uses frequent token sequence and flag bugs by recognizing deviations in mined patterns for renaming variables when copying-and-pasting the code, whereas Wahler *et al.* (2004) approach find exact and parameterized clones at a more abstract level by converting the AST to XML by using frequent item set-mining technique.

The development of high quality software within a reasonable time scale is also desired by software industry. Software reusability by reusing the existing software artifacts is one of the solutions to achieve this objective. Most of work that has been done in reusability area is identifying related component from reusable libraries. Over time components are added and it does not take so long to grow libraries to enormous proportion and complications related to reuse components are raised. Capretz (2004) proposed a classification scheme for arranging, selecting and reusing the

desired components from existing libraries. A literature review conducted on software reusability by Fazal-e-Amin *et al.* (2011) shows 60% of approaches using open source software repositories to develop new system. Researchers also mine data from software repositories in context of reuse of component. Different mining techniques proposed in the literature to retrieve samples code from the example repositories. Each techniques is differ in the means how developer uses these repositories to retrieve relevant code examples. For example, Strathcona (Holmes and Murphy, 2005) use structural context to automatically extract query from the code. Xsnippets (Sahavechaphan and Claypool, 2006) uses class structure information such as parents, fields and methods of a class to define code context to query a sample repository for code snippets. Prospector (Mandelin *et al.*, 2005), Parseweb (Thummalapenta and Xie, 2007) and MAPO (Xie and Pei, 2006) defines a query that describes the desired code.

All the studies mention above mines individual pattern types alone to accomplish a certain SE task. Thus, programmers need to employ multiple methods to mine different kind of useful information to improve software systems. However, SE tasks increasingly demand the mining of multiple correlated patterns together to achieve the most effective result. In this study an integrated framework for extraction of multiple patterns from program source code by using association rule mining and frequent subsequence mining aided by constraint-based mining is proposed. By proposing a novel layered architecture to adaptively perform multiple data mining techniques at different layers it is able to extract different types of correlated patterns to achieve the most effective results. The framework extracts a variety of patterns from source code which assists in improvement of multiple software engineering tasks over different phases of development life cycle e.g., assisting programming in writing code, bug detection and software maintenance.

Proposed approach consists of three phases. First, user specifies threshold and collects the implicit programming rules, copy pasted code segment as well as frequently occurred variable correlations. Second, user specifies the constraints to be imposed on searched rules to prune search space according to given conditions. The output contains all the relationships that satisfy the given constraints. Third, it also automatically scans the source code to detect all instances where given rules are violated, correlated variables are not updated consistently and copy-paste code related bugs. The main objectives of conducting the study are:

- Integrated pattern mining framework
- The introduction of a novel framework for constraint-based mining of source code
- The introduction of a number of interesting constraints for mining source code
- Development of interactive tool which support following activities:
    - User defines the attributes/constraints of rule to be mined through an interactive GUI
    - Automatically identifying function pairing rules in large program
    - Identifying the commonly existent multi-variable access correlations
    - Locating copy-paste code segments
    - By querying the open source repositories identifying the set of APIs that use to program the task at hand
    - Searching for all occurrences of a given rules that were used to program a piece of source code
    - Finally, scan the source code and searching for lines of code which violates the mined rules
- Initial empirical results characterizing the effectiveness of our approach

RELATED WORK

Abundant of tools and techniques are proposed in literature which extracts knowledge from source code and data mining methods are used for different purpose. In recent studies dedicated to the research of static source code mining four common areas of interest are identified namely, Rules Mining Techniques, Mining Variable Correlations, Detecting Copy-Paste Code and API Usage.

Engler *et al.* (2001) has conducted a preliminary investigation in direction of Mining Rules from source code. Proposed approach mines function's pairing rules by using compiler extensions called checkers to match programmer specified rule templates. It extracts programming beliefs from acts at different location of source code. Since approach relies on developers to supply rule templates such as function A must be paired with function B and covers the given or explicit rules known in advance, it may miss many violations due to the existence of implicit rules. PR-Miner developed by Li and Zhou (2005) find implicit programming rules and rule violations that is based on frequent item-set mining and does not require specification of rule templates. It can detect simple function pair-wise rules, complex rules as well as variable correlation rules. It computes the association in entire program elements by just counting the together occurrences of any two elements and not considering data flow or control flow which leads to increase number of false negative of violations in control path. CHRONICLER developed by Ramanathan *et al.* (2007) applies inter-procedural path-sensitive static analysis to automatically infer accurate function precedence protocols which specify ordering among function calls. CHRONICLER fundamentally differs from PR-Miner as it ensures path-sensitivity hence generate less number of false negative. Chang *et al.* (2007) proposed a new approach to mine implicit condition rules and to detect neglected conditions by applying frequent sub graph mining. The approach requires the user to indicate minimal constraints on the context of the rules to be sought, rather than specific rule templates. However, frequent sub-graph mining algorithm does not handle directed graphs and multigraphs and require the modification leads to information loss so that precision is sacrificed in rule discovery. Another approach developed by Lu *et al.* (2007) called MUVI to mine variable pairing rules which applied the frequent itemset mining technique to automatically detect multi-variable inconsistent update bugs and multi-variable related concurrency bugs which may result due to inconsistent update of correlated variables. Engler *et al.* (2001) work also detect variable inconsistency through logical reasoning where as MUVI (Lu *et al.*, 2007) detect inconsistencies using pattern analysis on multi-variable access correlations.

Several automated techniques for detecting code clones have been proposed differ by the level of comparison unit from single source lines to entire AST/PDG sub-trees/sub-graphs. However, we only focus on techniques which use data mining and few others leading techniques for clone detection such as CCFinder (Kamiya *et al.*, 2002) and Dup (Baker, 1995) that use tokenization on the source code. Dup detect two types of matching code that is either exactly the same or name of parameters such as variable and constant are substituted. CCFinder detect clone code portions that have different syntax but have similar meaning and applies rule-based transformation such as regularization of identifiers, identification of structures, context information and parameter replacement of the sequence. Abstract syntax tree based approaches (Wahler *et al.*, 2004) and PGDs based (Qu *et al.*, 2010) tools looks for sub trees and isomorphic graphs to find clones. In addition to above and many other technique we find only two approaches that, CP-Miner (Li *et al.*, 2004) and Clonedetection (Wahler *et al.*, 2004) which uses data mining to detect clones. CP-Miner uses frequent token sequence and flag bugs by recognizing deviations in mined patterns for renaming variables when copy-and-pasting the code. It transforms a basic block into number

by tokenizing its component. Once all the components of a statement are tokenized, a hash value digest is computed using the hashpjw hash function. The ColSpan algorithm is applied to the resulting sequence database to find basic copy-pasted segments. By identifying abnormal mapping of identifiers among copy-paste segments, CP-Miner detects copy-paste related bugs, especially those bugs caused by the fact that the programmer forgot to modify identifiers consistently after copy-pasting. Whereas, Wahler *et al.* (2004) approach find exact and parameterized clones at a more abstract level by converting the AST to XML by using frequent item set-mining technique. This tool first converts source code into Abstract Syntax Tree (AST) which contains complete information about source code by using parser. Frequent itemset mining algorithm inputs XML configuration file and find frequent consecutive statements. Proposed technique only finds exact and parameterized clones at a more abstract level.

Much research has been conducted to extract API usage rules or patterns from source code by proposing tools and approaches which helps developers to reuse existing frameworks and libraries more easily. Prospector developed by Mandelin *et al.* (2005) automatically synthesize the list of candidate jungloid code based on simple query that described the required code in term of input and output. The Jungloid graph is created using both API method signatures and a corpus of sample client programs and consists of chains of objects connected via method calls. Prospector mines signature graphs generated from API specifications and jungloid graphs. The retrieval is accomplished by traversing a set of paths (API method call sequences) from T_{in} to T_{out}. The code snippets returned by this traversal process are ranked using the length of the paths with the shortest path ranked first from Tin to Tout.

MAPO developed by Xie and Pei (2006), mines frequent usage patterns of API through class inheritance. It uses API's usage history to identify methods call in the form of frequent subsequences. The code search engine receives a query that describes a method, class or package for an API and then searches open source repositories for source files that are relevant to the query. The code analyzer analyzes the relevant source files and produces a set of method call sequences. The sequence preprocessor inline some call sequences into others based on caller-callee relationships and removes some irrelevant call sequences from the set of call sequences according to the given query. The frequent-sequence miner discovers frequent sequences from the preprocessed sequences. The frequent-sequence postprocessor reduces the set of frequent sequences in some ways.

Sahavechaphan and Claypool (2006) developed a context-sensitive code assistant tool XSnippet, that allows developers to query for relevant code snippets from a sample code repository to find code fragments relevant to the programming task at hand. A range of instantiation queries are invoked from java editor including generic query TQ_G that returns all possible code snippets for the instantiation of a type, to the specialized type-based TQ_T and parent based queries TQ_P that return either type-relevant or parent-relevant results. User input the type of query, code context in which query is invoked and a specific code model instance to graph based Xsnippet system. Mining algorithm BFSMINE, a breath first mining algorithm traverses a code model instance and produces as output that represent the final code snippets meet the requirement of the specified query.

PARSEWeb developed by Thummalapenta and Xie (2007), uses Google code search for collecting relevant code snippets and mines the returned code snippets to find solution jungloids. The proposed technique described the desired code in the form of source destination query which search for relevant code sample of source and destination object and download to form a local source code repository which is analyzed to constructs a directed acyclic graph. PARSEWeb identifies nodes

that contain the given source and destination object types and extracts a Method-Invocation Sequences (MISs). PARSEWeb clusters similar MISs using a sequence postprocessor. The final MISs are sorts using several ranking heuristic and serves as a solution for the given query. PARSEWeb also uses an additional heuristic called query splitting that helps address the problem where code samples for the given query are split among different source files.

CONSTRAINT BASED MINING FRAMEWORK

To automatically mine patterns from source code, a novel framework s developed which identify implicit programming rules, copy-paste code segments, multivariable access correlation and detect rule violating defects, copy-paste related defects and multi-variable related inconsistent updates defects. Moreover, the framework also provides the facility to developer to get a short list of relevant frequent API usage pattern. The study combines the static program analysis and data mining techniques to find the rules and their violation. It uses the frequent item set mining to find programming rules, variable pairing rules and frequent subsequence mining to find copy-paste code and API usage pattern. Figure 1 show the abstract model of framework.

Finding patterns/rules: It consists of four major components. Code Generator/analyzer, rule mining engine, sequence miner and constraint based miner.

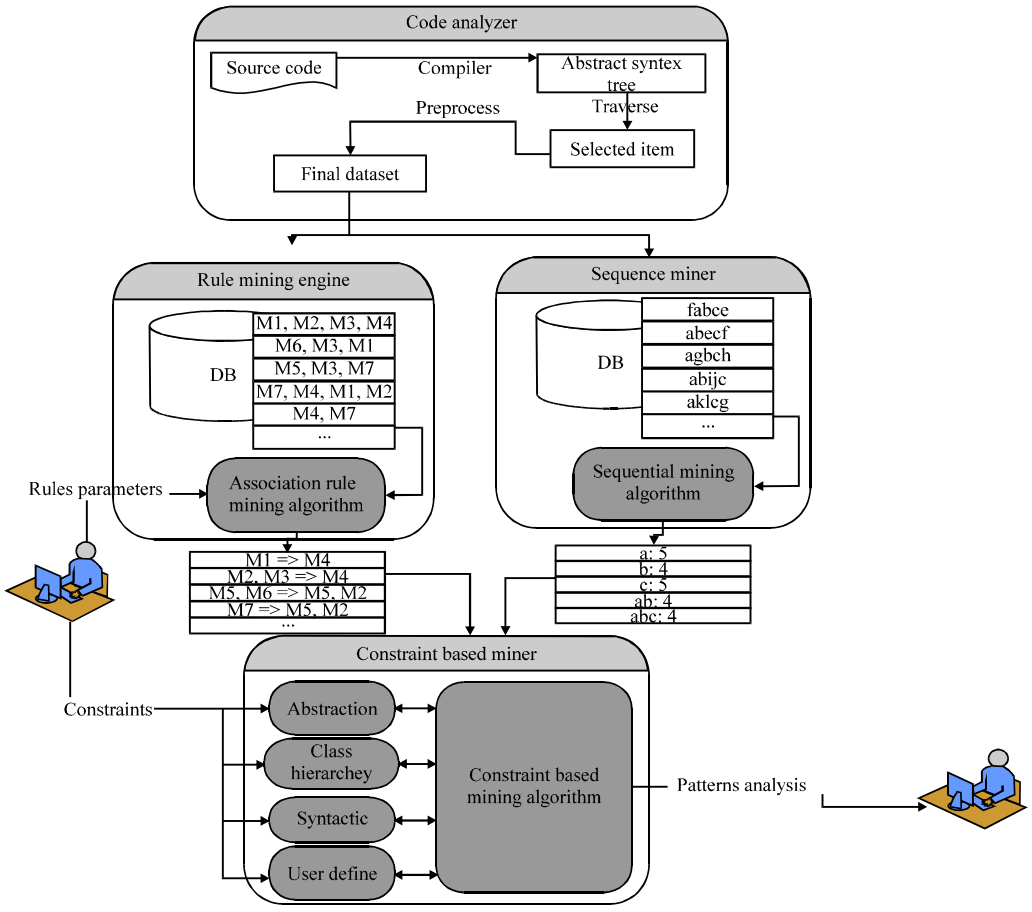

Fig. 1: Abstract model of framework

Code analyzer: Code Analyzer parses the software system and generates a database of system entities and their relationship by using the GCC compiler (Stallman, 2002). The main task is transformation of source code into database suitable to apply mining algorithm. The intermediate representation of source code is stored in a tree structure. An abstract tree represents a function, where the leaves and the internal nodes denote different types of elements in the function. Each abstract tree is reversed to select the items, such as identifier names, data type specifiers, operators and control structures, Hence, function is converted to an itemset. The process of code analyzer is shown in Fig. 1.

Rule Mining Engine (RME): Rule Mining Engine receives transaction database as an input and generates a number of association rules by applying Association Rule Mining algorithm Apriori (Agrawal and Srikant, 1994). A sub-itemset (a subset of an itemset) is considered frequent if the number of its occurrences in the database, denoted as its support, is greater than or equal to a specified threshold (called the minimum support). Items in a frequent pattern are likely to have some correlation in between. For a frequent sub-itemset discovered by the mining algorithm, the set of the corresponding program elements call as programming pattern which indicates that the program elements are correlated and frequently used together. For example, Let I be the set of all items present in database. Rule mining engine search for the power set of I for each patterns classes satisfying minimum threshold.

$$I = \{\{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\}\}$$

The support of sub-itemset {a, b, d} is 3 and its supporting itemsets are {a, b, c, d, e}, {a, b, d, e, f} and {a, b, d, g}, If min support is specified as 3, the frequent sub-itemsets for I are {a}: 4, {b}: 3, {d}: 3, {a, b}: 3, {a, d}: 3, {b, d}: 3 and {a, b, d}: 3, where the numbers are the supports of the corresponding sub-itemsets. The location (where the items locate in the source code) of each frequent itemsets is also recorded which is required when evaluating if a violation is a defect. Once all patterns Z are identified, all rules are generated by splitting Z in head X and body Y such that $X \cup Y = Z$ and $X \Rightarrow Y$ with support s and confidence c satisfy the rule. For each pattern class set of rules are generated. Save the resulting pattern Z and rules which serve an input to second phase. RME generates set of all rules showing item which frequently occurred together in source code such as function pairing as well as variable correlation rules.

Sequence miner: Since the items in the sequence database are ordered by their associated order. All items in an itemset are assumed to occur at the same time. A sequence is an ordered list of itemsets. Order is added to every item in each transaction converted from the source code. Here, line number of a function or a variable is used as order. Frequent sequence mining is to discover all the subsequences occur frequently in a given database, beyond a user specified min_sup threshold. Prefixscan (Pei *et al.*, 2001) is applied on sequence database to find copy paste code segment and API usage pattern.

Constraint-based miner: The set of all patterns holding in the source code data provide a wealth of information. A given relation may satisfy a large number of association which reveals valuable

unexpected information. This strength of association rule has drawback: There can be quite many rules holding with sufficient strength in data set. In the proposed technique this large number of generated information are managed by offering constraint based mining which enable the users to clearly specify which associations are to be mined during the extraction process. This enables to reduce the number of generated association rules and, usually, to increase the quality of the extracted model. Constrain based miner includes following steps:

- **Rule selection:** The first component is the subjects for evaluation of discovered rules which is what needs to be evaluated. Rule selection is a tool for specifying the condition for rule from user. In this component user defines the parameter of a rule which serve as a constraint for rule to be evaluated
- **Process rules:** This Component process rules according to given conditions. Rule search space is prune by exploiting following properties of constraints
    - **Abstraction constraints:** It defines a generalization of some items on a concept of the class hierarchy by exploiting the Abstraction Constraints category. Attributes are classified to inheritance class hierarchy by the user on the basis of existing knowledge. Only the pattern matches with given class instance are searched by rule
    - **Hierarchy relations constraint (comparable to monotone constraint):** One of the properties of the classes analyze is the class hierarchy relationship. This causes our algorithm to identify implication rules that express mere subclass relationships (e.g., if a class is in the hierarchy of class A, it also is in the hierarchy of class B is trivially true for each super class B of A). Since such implication rules present trivial information, they are pruned from the final result
    - **Syntactic constraints:** On items that appear in a rule are put restrictions which concerns whether it should be placed as antecedent or consequent. The rules that have item X as an antecedent and item Y as a consequent can be searched. Of course X and Y can even have set of items but they have to be disjoint

Finding rules violations: This step in proposed approach is finding violations of the rules that were identified by frequent itemset and sequence mining and confirmed by the user. The process of finding violation is shown in Fig. 2. Rules are provided in the way that two functions must be use in source code in giving sequence. If given sequence is not followed in source code it violates the rule and prototype shows the results as rule violation. First the source code is transformed into tokens and hash value is assigned to each token. Each value is matched with token by token with given rule. For loop is used for matching token by token with given rules, each token string is matched with first rule if matching found it stores in 2-D array as well as token string an its location/Index. If one rule is not match with token then token is match with 2nd Input rule. if match found then store into 2-D array type result. It repeats until all rules found. For finding the variable access correlation for loop is used for checking one by one from token string. If matching variables found they stores in long data types. The process repeats until last token is checked. At the end results are shown in output box. Once all rules and its locations are found we find the violation where given sequence is not followed.

PRELIMINARY RESULTS

To validate the proposed framework a prototype is developed which can accepts source code of any language and determine the implicit programming rules, variable correlation and identify copy paste code. The prototype also provides the facility to input a query and provide the reusable API code from web and open source code repositories. Once desired patterns are obtained the prototype also checks for violation of patterns and locate the location of source code where desired standard is not followed. Table 1 shows the list of the hardware and software used for developing prototype. Table 2 shows the system detail on which proposed system is evaluated. Table 3 and 4 show the sample of initial empirical results by showing example line of code on which initially rule is found and lines of code on which rules were violated. Figure 3 shows the frequency of each type of pattern found by running the prototype on Metro HRM Solution. Figure 4-7 show the results in image snippet of the application developed on the software and hardware support as well as system evaluated mentioned in Table 1 and 2. Although, the program developed on the proposed architecture is prototype, still focus on reasonable easiness and better usability has given prior importance.

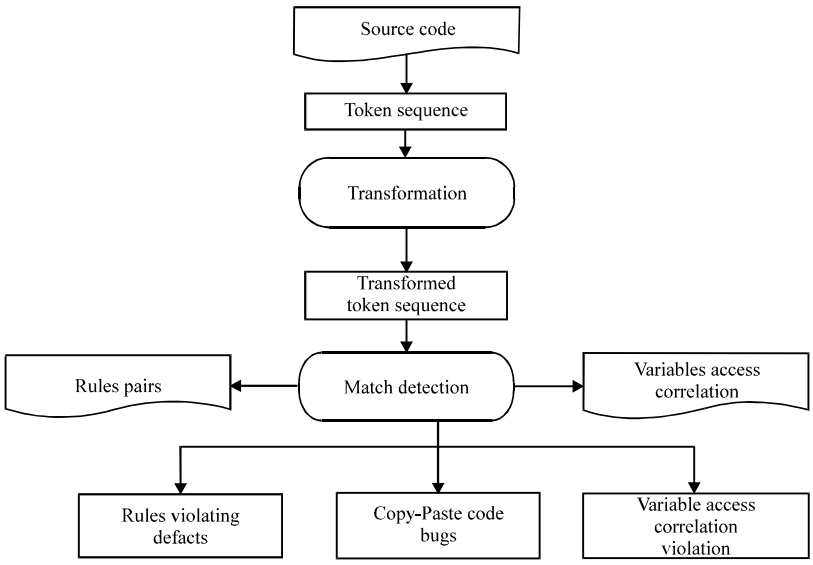

Fig. 2: Finding rules violations

Table 1: Software and hardware support detail

Operating system	Windows XP
Programming language	C# (MS Visual Studio 2008)
Databases	Oracle 10 g
PC hardware specification	Pentium Intel 3.2 GHz/1GB Ram

Table 2: System evaluated

No.	System: Metro HRM solution
1	Source code: C# sample code (1.22 GB)
2	789 No. of files
3	1335 No. of functions
4	Average results 10 times
5	Oracle 10 g, Database (18 GB)

Table 3: Rule violation results

Rule identified	Rule violation found	Description
50-51	681, 946	objData.CloseConnection ();
		objData.disposeConnection ();
233-234	345, 346	openConnection ();
		sqlTrans =this Connection. Begin Transaction ();

Table 4: Copy-paste code results

Code segment	Copy-paste code identified	Copy-paste bugs
50-55	59, 73, 681, 946	Line 681
86-92	174, 212, 245, 247, 265, 640, 981	Line 121, 640
117-130	124, 134, 145, 156, 167, 182, 190, 198, 206, 222, 231, 239	Lines 190, 222, 239

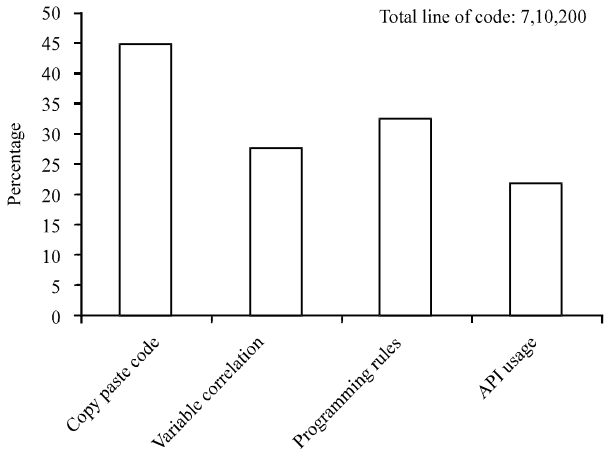

Fig. 3: Overall system results in four dimension

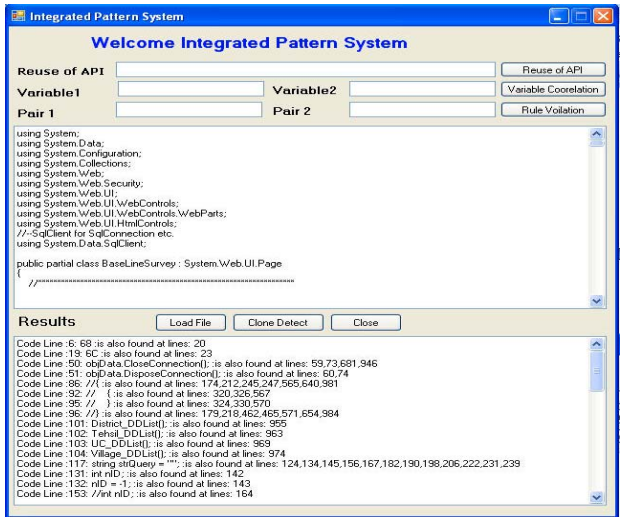

Fig. 4: Copy-paste code results

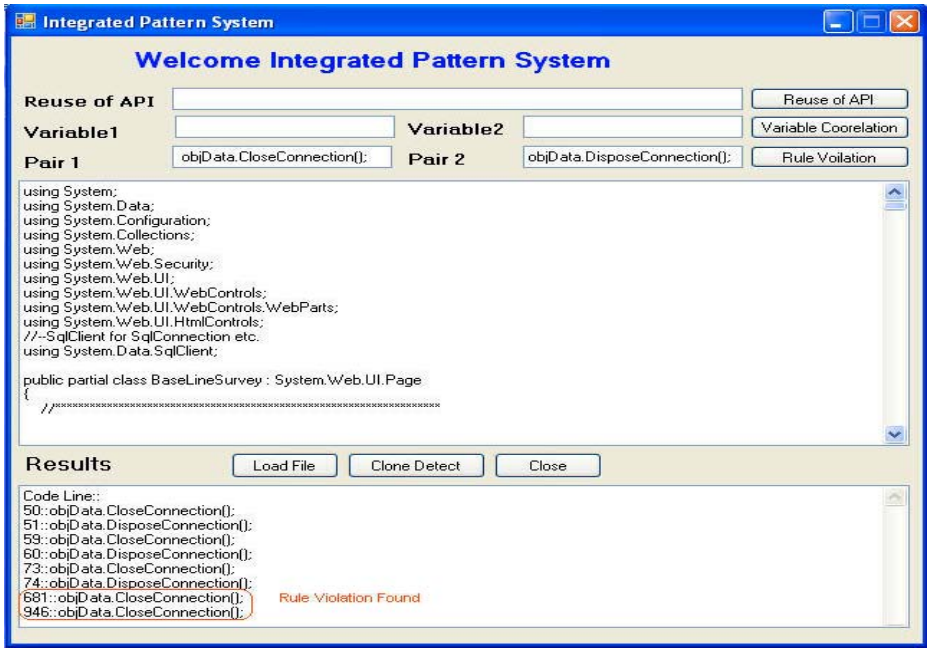

Fig. 5: Rules violation results

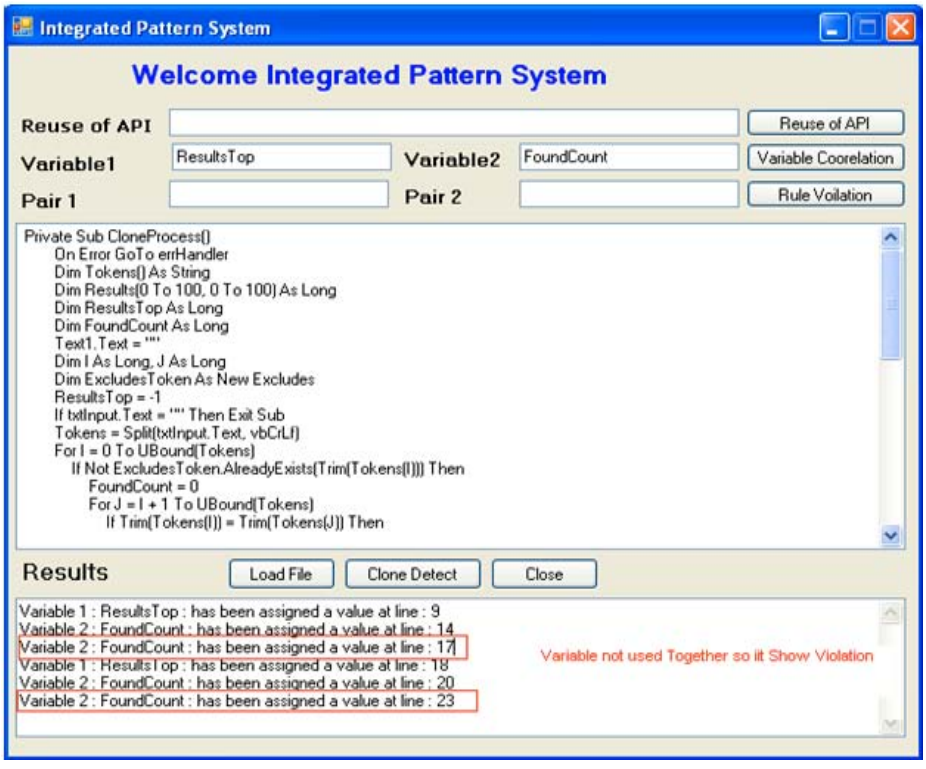

Fig. 6: Variable correlation violation results

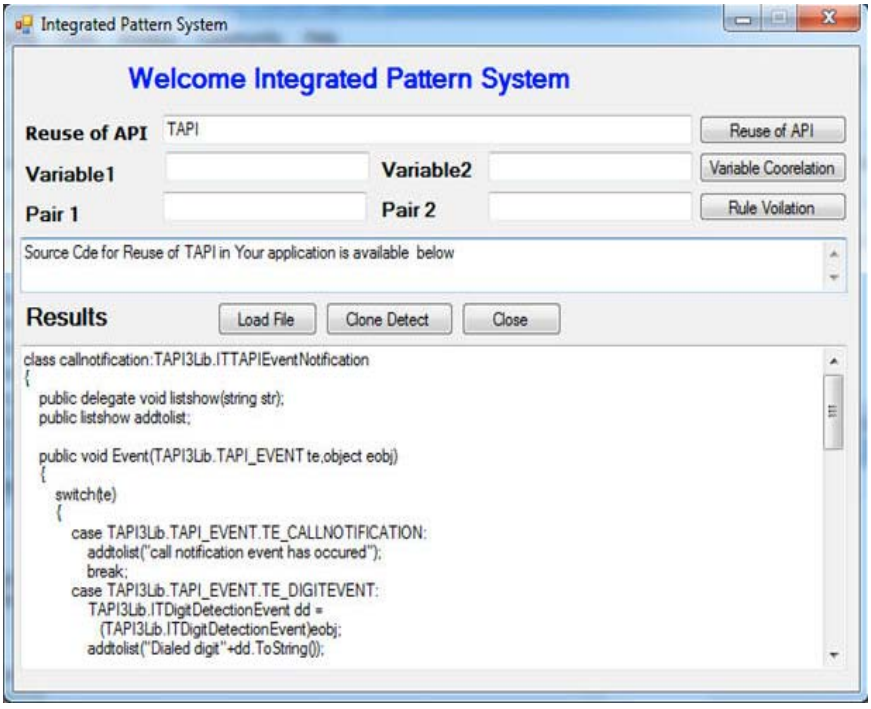

Fig. 7: API usage results

CONCLUSION

In this study an integrated framework is developed which can automatically find all the patterns from source code in one pass and suggest potential bug locations for quality software development and relevant code suggestion for rapid development. So far no such kind of integrated framework is available which can find multiple patterns and various kinds of bugs in one pass and reduces time and cost for software development and testing as proposed method is. In addition, this method is able to mine implicit programming rules, variable correlations and copy-paste segments in software requiring little prior knowledge from programmers. Novel constraint based techniques have also been proposed to prune uninteresting rules in rule mining and sequence mining process. In future, development of fully automated system is planed which accept any kind of source file, extract patterns and detect bugs as well as suggest relevant code.

REFERENCES

Agrawal, R. and R. Srikant, 1994. Fast algorithms for mining association rules. Proceedings of the 20th International Conference on Very Large Data Bases, Sept. 12-15, San Francisco, CA., USA., pp: 487-499.

Alsmadi, I., 2011. Activities and trends in testing graphical user interfaces automatically. J. Software Eng., 5: 1-19.

Baker, B.S., 1995. On finding duplication and near-duplication in large software systems. Proceedings of the 2nd IEEE Working Conference on Reverse Engineering, July 14-16, IEEE Computer Society, Toronto, Canada, pp: 86-95.

Baxter, I.D., A. Yahin, L. Moura, M. Sant'Anna and L. Bier, 1998. Clone detection using abstract syntax trees. Proceedings of the International Conference on Software Maintenance, Nov. 16-20, Bethesda, MD., USA., pp: 368-377.

Capretz, L.F., 2004. A software process model for component-based development. Inform. Technol. J., 3: 176-183.

Chang, R.Y., A. Podgurski and J. Yang, 2007. Finding what's not there: A new approach to revealing neglected conditions in software. Proceedings of the International Symposium on Software Testing and Analysis, (STA'07), ACM Press, New York, pp: 163-173.

Engler, D., D. Chen, S. Hallem, A. Chou and B. Chelf, 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. ACM SIGOPS Oper. Syst. Rev., 35: 57-72.

Fazal-e-Amin, A.K. Mahmood and A. Oxley, 2011. A review of software component reusability assessment approaches. Res. J. Inform. Technol., 3: 1-11.

Hassan, A.E. and T. Xie, 2010. Mining software engineering data. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, (ICSE'10), ACM Press, New York, pp: 503-504.

Holmes, R. and G.C. Murphy, 2005. Using structural context to recommend source code examples. Proceedings of the 27th International Conference on Software Engineering, May 15-21, ACM Press, New York, pp: 117-125.

Kamiya, T., S. Kusumoto and K. Inoue, 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Eng., 28: 654-670.

Khoshgoftaar, T.M., A.S. Pandya and D.L. Lanning, 1995. Application of neural networks for predicting program faults. Ann. Software Eng., 1: 141-145.

Kosindrdecha, N. and J. Daengdej, 2010. A test case generation process and technique. J. Software Eng., 4: 265-287.

Krinke, J., 2001. Identifying similar code with program dependence graphs. Proceedings of the 8th Working Conference on Reverse Engineering, Oct. 2-5, IEEE Computer Society, Stuttgart, Germany, pp: 301-309.

Li, Z. and Y. Zhou, 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Sept. 5-9, Lisbon, Portugal, pp: 306-315.

Li, Z., S. Lu, S. Myagmar and Y. Zhou, 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation, (SOSDI'04), San Francisco, CA., pp: 289-302.

Lu, S., S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa and Y. Zhou, 2007. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. ACM SIGOPS Oper. Syst. Rev., 41: 103-116.

Lu, Y. and M. Ye, 2007. Oracle model based on RBF neural networks for automated software testing. Inform. Technol. J., 6: 469-474.

Maamri, R. and Z. Sahnoun, 2007. Multi-agent plateform for software testing. Inform. Technol. J., 6: 48-56.

Mandelin, D., L. Xu, R. Bodik and D. Kimelman, 2005. Jungloid mining: Helping to navigate the API jungle. ACM SIGPLAN Not., 40: 48-61.

Pedrycz, W. and J.F. Peters, 1997. Computational intelligence in software engineering. Proceedings of the Canadian Conference on Electrical and Computer Engineering, (CCECE'97), IEEE, pp: 253-256.

Pei, J., J. Han, M.A. Behzad, P. Helen, Q. Chen and M.C. Hsu, 2001. Prefix Span: Mining sequential patterns efficiently by prefix-projected pattern growth. Proceedings of the 17th International Conference on Data Engineering, (ICDE'01), Heidelberg, Germany, pp: 215-224.

Qu, W., Y. Jia and M. Jiang, 2010. Pattern mining of cloned codes in software systems. Inform. Sci., (In Press).

Ramanathan, M.K., A. Grama and S. Jagannathan, 2007. Path-sensitive inference of function precedence protocols. Proceedings of the 29th International Conference on Software Engineering, May 20-26, Minneapolis, MN., USA., pp: 240-250.

Roongruangsuwan, S. and J. Daengdej, 2010. A test case prioritization method with practical weight factors. J. Software Eng., 4: 193-214.

Sahavechaphan, N. and K. Claypool, 2006. XSnippet: Mining for sample code. ACM SIGPLAN Not., 41: 413-430.

Shu, Y., H. Liu, Z. Wu and X. Yang, 2009. Modeling of software fault detection and correction processes based on the correction lag. Inform. Technol. J., 8: 735-742.

Stallman, R.M., 2002. GNU compiler collection internals. Free Software Foundation, http://www.gnuarm.com/pdf/gccint.pdf

Thummalapenta, S. and T. Xie, 2007. Parseweb: A programmer assistant for reusing open source code on the web. Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Nov. 4-9, Atlanta, Georgia, USA., pp: 204-213.

Vinayagasundaram, B. and S.K. Srivatsa, 2007. Software quality in artificial intelligence system. Inform. Technol. J., 6: 835-842.

Wahler, V., D. Seipel, J.W.V. Gudenberg and G. Fischer, 2004. Clone detection in source code by frequent itemset techniques. Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, Sept. 16, IEEE Computer Society, Chicago, IL., USA., pp: 128-135.

Xie, T. and J. Pei, 2006. MAPO: Mining API usages from open source repositories. Proceedings of the 2006 International Workshop on Mining Software Repositories, May 22-23, Shanghai, China, pp: 54-57.