Detecting of Software Source Code Defects using Test Case Prioritization Rules

T.Prem Jacob ¹, and Dr.T.Ravi ²

Abstract--The complexity and large size of software source code possibly cause defects in software results in causing the program to freeze or crash while execution. So an effective defect detection system is required to analyze the source code and to eliminate the defects. This project applies data mining techniques to detect two types of defects in source code in single process. The two types of defects are as follows: Rule-violating defects and Copy-paste defects. An example for Rule violating defect is missing of function braces (Lock and unlock). Copy paste defect includes unfortunately adding extra spaces in source code while doing copy-paste activity. The proposed defect detection method can extract implicit programming rules of source code without any prior knowledge about it. This method first uses Frequent Pattern mining (Apriori) to detect rule-violation defects then uses Frequent Sequence Mining to detect copy-paste defects. The first step in detection process is transformation of functions present in source code into some format of Dataset. Then the rule-violating defect detection and Copy paste defect detection methods are applied on the data set in one step. Finally the defect report is generated. Hence the proposed frequent pattern mining and frequent sequence mining techniques reduce manual effort that is need for defect detection.

Keywords-  Rule violating, copy-paste, software source code, defect detection.

I. INTRODUCTION

With the rapid development of software industry, the size and complexity of software is increasing drastically. As a result, more and more defects emerge in software. Serious defects can cause the programs either to freeze or to crash, which may result in a denial of service. A study is concluded which was specially made by the US Department of Commerce National Institute of Standards & Technology in 2002 stating that the software errors or software bus are very common and very harmful where they cost an estimated of about $59 billion per annum for the US economy or about 0.6% of the gross domestic product. This work mainly focus on two types of defects: rule violating defects and copy-paste related defects, most of which are semantic defects.

A. Rule-violating defect

Some violations of implicit programming rules are evident. For example, function pair lock and unlock must be used together, that is, a call to lock should be followed by a call to unlock.

B. Copy-paste defect

Copy-paste exists in the source code of many software programs. Recent studies showed that a great portion of code is reused. For example, CCFinder a copy-paste detection tool, found that 12% of the Linux file system code (279K lines) was involved in code cloning activity. Baker found that 19% of the source of X Window system (714K lines) code was duplicates. Although copy-paste may cause problem, many programmers still tend to use it since can save time and increase productivity. This paper proposes a defect detection method which can detect the two types of defects. Proposed method can detect both rule-violating defects and copy-paste related defects in one pass. Existing defect detecting methods usually focus on only one specific type of defect. Thus, programmers need to employ multiple methods to test their software. Applying our method can improve the efficiency of software testing. Proposed method can reduce the number of false positives by using our novel pruning techniques. Examining the false positives may waste programmers’ precious time. By introducing our pruning methods, the number of false positives is reduced significantly.

II. RELATED WORK

This section takes survey on earlier work with different kinds of techniques. The Programs usually follow many implicit programming rules, most of which are too tedious to be documented by programmers. When these rules are violated by programmers who are unaware of or forget about them, defects can be easily introduced. Therefore, it is highly desirable to have tools to automatically extract such rules and also to automatically detect violations. Previous work in this direction focuses on simple function-pair based programming rules and additionally requires programmers to provide rule templates. This paper proposes a general method called Apriori that uses a data mining technique called frequent itemset mining to efficiently extract implicit programming rules from large software code written in an industrial programming language such as C, requiring little effort from programmers and no prior knowledge of the software.
Benefiting from frequent itemset mining, Apriori can extract programming rules in general forms (without being constrained by any fixed rule templates) that can contain multiple program elements of various types such as functions, variables and data types. In addition, we also propose an efficient algorithm to automatically detect violations to the extracted programming rules, which are strong indications of bugs. Our evaluation with large software code, including Linux, PostgreSQL Server and the Apache HTTP Server, with 84K–3M lines of code each, shows that Apriori can efficiently extract thousands of general programming rules and detect violations within 2 minutes. Moreover, Apriori has detected many violations to the extracted rules. Among the top 60 violations reported by Apriori 16 have been confirmed as bugs in the latest version of Linux, 6 in PostgreSQL and 1 in Apache. Most of them violate complex programming rules that contain more than 2 elements and are thereby difficult for previous tools to detect. We reported these bugs and they are currently being fixed by developers. [1] In a real system, the main problem to detect the program errors identifies the exactness rules of the system and it will obey. These rules are précised in an Adhoc manner and also undocumented in an Adhoc manner. The proposed approach which does not require any knowledge if two beliefs contradict, we know that one is an error without knowing what the correct belief is. Conceptually, our checkers extract beliefs by tailoring rule "templates" to a system - for example, finding all functions that fit the rule template "<a> must be paired with < b>." 

In this paper, 6 checkers are developed which follows this theoretical framework where they can identify 100’s of errors or bugs like OpenBSD & like Linux in the real system. From the earlier experience it shows the vivid reduction in the manual attempt which required to check the large systems. These template checkers can identify 10 to 100 times more rule cases and also it can obtain properties which was found impossibly to state manually. [2] This paper describes LCLint, an efficient and flexible tool that accepts as input programs (written in ANSI C) and various levels of formal specification. Using this information, LCLint reports inconsistencies between a program and its specification. We also describe our experience using LCLint to help understand, document, and re-engineer legacy code. [3] We present pSPADE, a parallel algorithm for fast discovery of frequent sequences in large databases. pSPADE decomposes the original search space into smaller suffix-based classes. Each class can be solved in main-memory using efficient search techniques and simple join operations. Furthermore, each class can be solved independently on each processor requiring no synchronization. However, dynamic interclass and intraclass load balancing must be exploited to ensure that each processor gets an equal amount of work. Experiments on a 12 processor SGI Origin 2000 shared memory system show good speedup and excellent scaleup results. [4]

III. PROPOSED WORK

In this paper, the system employs a frequent pattern mining algorithm to detect the rules violation defects, and then with the use of the frequent sequence mining algorithm it also finds out the defects like the copy-paste related defects. These methods are been applied on the data set which is been preprocessed data set (i.e Preprocessed source code) on the same time interval. To decrease the false positives number, a false positive values pruning techniques are been employed to it. Also the proposed approach reduces the time and the cost for software testing. We propose a general method to automatically extract implicit programming rules from large software code. Benefiting from data mining techniques, Apriori can extract thousands of programming rules from software such as Linux with 3500 files and a total of 3 million lines of code within 1 minute. Compared with the previous work [8] that extracts only function-pair based rules, Apriori extracts substantially more rules. This is because Apriori has substantially generalized Engler et al’s work in the following two aspects:

A. Normal Thechnique

The normal technique for extracting programming rules is more general. Apriori can automatically extract programming rules from software code without any prior knowledge about the software or requiring any annotation, templates or weight assignments from programmers. Additionally, by replacing the frontend parser, Apriori can be easily modified to work with programs written in other programming languages such as Java.

B. Common rules

The programming rules extracted by Apriori are more general. Since it does not limit the programming rules using any fixed templates, Apriori can extract rules in general forms and with multiple program elements of different types including functions, variables, data types, etc. As a result, it not only extracts simple pair-wise rules, but also extracts more complex rules like the examples shown above. We also propose an efficient algorithm to detect violations to the extracted programming rules. Apriori has detected many violations to the extracted rules in the latest versions of Linux and PostgreSQL within 1 minute. Among the top 60 violations reported by Apriori, many of them have been confirmed as bugs, including 16 bugs in Linux, 6 bugs in PostgreSQL and 1 bug in Apache. These bugs are currently being fixed by corresponding developers after we reported. Most of these bugs are semantic bugs that violate complex programming rules that contain more than 2 elements and are thereby difficult for previous tools to detect.

The input of our system is given by the user. The input comprises the general information of particular domain provided by the user. The domain provided by the user can be of a testing application of a java based program module. The selection of the criteria is done by attributes of the different test case conditions. The attributes selected are also influences the selection of the algorithm to be used for mining.
C. A-priori algorithm

Uses a Level-wise search, where k-itemsets (An itemset that contains k items is a k-itemset) are used to explore (k+1)-itemsets, to mine frequent itemsets from transactional database for Boolean association rules. First, the set of frequent 1-itemsets is found. This set is denoted L1. L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-itemsets can be found. The apriori algorithm is an efficient algorithm for knowledge mining in form of association rules [2]. We have recognized its convenience for document categorization. The original apriori algorithm is applied to a transactional database of market baskets. In our case, instead of a market basket, we work with the testing and test case selection (represented by sets of significant terms).

Using closed frequent itemset mining algorithms such as FPclose provides Appriori several benefits:

D. Generality

Close frequent itemset mining algorithms do not limit the number of items in frequent sub-itemsets and also does not require any rule templates. Furthermore, the items in a frequent sub-itemset are not necessarily adjacent in the supporting itemset, i.e. they can be far apart from each other.

E. Time efficiency

Data mining algorithms such as FPclose are usually very efficient since they strive to avoid scanning data too many times by eliminating redundant computation as much as possible.

F. Space efficiency

From closed frequent sub-itemsets, we can find closed rules, rules that subsume many other rules with the same support. Using closed rules not only saves space, but also significantly reduces the number of rules that need to be examined or referenced by programmers. In addition, it also speeds up the violation detection process.

IV. DETECT RULES

The main idea is that the programming rules usually hold for most cases and violations happen only occasionally. Take the potential bug detected by Apriori. The function call to scan should follow alloc and add as the programming rule indicates. This rule appears 27 times in Linux, but there are 2 cases violating this rule because scan is missing. The system first detects violations to the extracted programming rules, then prunes the false violations using inter-procedural analysis, and finally ranks the violations in the error report.

A. Violation Detection

In order to detect violations to the programming rules, a naïve method is to generate all possible programming rules and then check them upon the source code one by one. First, if the confidence threshold is t, any rules with confidence smaller than t are discarded. Second, if the rule has 100% confidence, it indicates that there is no violation for the rule.

B. False Pruning

The violation detection above can result in false positives if the elements in a programming rule span across multiple functions. In order to check the call path backwards, Apriori maintains a caller list for each function that consists of the indexes of the functions that call F. If the missing items for function F are in the paths of all of its callers, it is a false violation.

C. Bug Reports

Apriori ranks the violations based on the confidence of the violated rules. Since a function may contain several violations, Apriori groups all violations of the same function together, and the violation with the highest confidence is assigned as the confidence of the violated function. In the current version of Apriori, it simply ranks the bugs by the confidence.

V. RESULTS

Our results show that a large number of implicit, undocumented programming rules can be effectively extracted from source code by apriori without any priori knowledge or annotations/specifications from programmers. For example, apriori extracts a total of 32,283 implicit, undocumented closed programming rules from Linux. It would be very difficult for programmers to manually specify these many programming rules. Apriori effectively relieves such burden from programmers by efficiently and automatically extracts such rules from source code.

The results also show that around 88.3–96.7% rules involve variables. For example, there are around 9000 V-V rules in Linux, along with a large number of variable correlations contained in F-V rules. Comparing with the previous studies such as Engler et al’s work [8] that do not consider rules about variable correlations, apriori can extract substantially more programming rules. The algorithm is selected based on the selected rules. Here input to the algorithm is the bit corresponding to each parameter. The input will be a row which contains the value of the parameter. For some
parameters the values may be kept as levels. The result explains about different the false alarm detection rate by different algorithms and is been shown in Figure 2.

![Figure 2. Different Algorithm Vs Accuracy of the False Detection rate](image)

Fig2. Different Algorithm Vs Accuracy of the False Detection rate

VI. CONCLUSION

Here it concludes that the system employs a frequent pattern mining algorithm to detect the rules violation defects, and then with the use of the frequent sequence mining algorithm it also finds out the defects like the copy-paste related defects. These methods are been applied on the data set which is been preprocessed data set (i.e Preprocessed source code) on the same time interval. To decrease the false positives number, a false positive values pruning techniques are been employed to it. Even though these kinds of defects are different the proposed approach uses the defect detection method which is easily identify the defects in one process. Also the proposed approach reduces the time and the cost for software testing. The approach will greatly reduce the effort of manually checking violations so as to generate less number of false positives.

REFERENCES

[4] Parallel Sequence Mining on Shared-Memory Machines Mohammed J. Zaki Computer Science Department, Rensselaer Polytechnic Institute, Troy, New York 12180