ISSN:0975-9646
Anil Patro,Raj Sekhar et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (4) , 2011, 1403-1406

A Technique for Identifying and Testing Structural Clones in Large Scale Systems

Anil Patro,Raj Sekhar,C.P.V.N.J.Mohan Rao
Department of CSE, Avanthi Institute of Engineering & Technology, Narsipatnam, Visakhapatnam district, Affiliated to J.N.T.U Kakinada, A.P, India.

***Abstract*-Clones have played an important role in increasing the software's maintenance and decreasing the quality. Hence detecting the clones and removing them has been an interesting scenario. Detecting clones not only improves the productivity of software but also enhances re-usability. A number of techniques were presented in the past but they have had there own drawbacks. Efficient algorithms are crucial for identifying structural clones. In this paper we present a tool that uses efficient data mining techniques like clustering and association rule mining. As a result we present some methods for detecting exact and near miss clones in program source code. We also present how to test the code using model-based testing techniques in order to test for the bugs at the time of running the tool. Our technique works very well for sparse datasets. Our technique is proposed to improve the speed of the clone detection. In order to reduce the number of comparisons required for clone detection, we select representative clones from the existing clone list by using manual techniques like tokenization.**

***Keywords*-Clones, Re-use, Clone-Detection, Maintenance, Structural Clones.**

I. INTRODUCTION

In large scale systems many projects are developed in parallel. The code of all the software projects is centrally stored in a system. Hence duplicated code is common in all kind of software systems. And also the data obtained from the previous work shows that a considerable amount of the source of large computer programs is duplicated code i.e., (10-20%) [2][3][4]. Clones are usually created by using adhoc copy-paste techniques and as a result extending the existing code. But due to this code duplication software maintenance has become more complicated. Detecting the lower level clones called simple clones has been easy but identifying the higher level clones i.e., clones at file and directory level has been an uphill task. It has been like discovering the trees through the forest.

Another important point ignored by many programmers is that the number of errors also will be duplicated together with the duplicated code. And as a result modifications made to the original version should also be applied to the duplicated code. If at all a new module is to be added to a system all the places related to it will require modifications. Hence it is difficult to assure the quality of the system. Whenever bugs are identified they should be fixed in all of the duplicated code. Failing to identify the duplicated code will increase the difficulty to fix the bugs. Hence it is important to locate clone data in large software and remove them as early as possible.

Previous clone detection [1] work was only limited to textual matches or near misses only on complete function bodies. Whereas this paper presents some practical methods for detecting exact and near miss clones for arbitrary fragments of program source code. And also the current clone detection approaches are not scalable to very large codes. Hence they cannot be used for real-time detection in large systems, thereby reducing their usefulness for clone management.

Hence in this paper we will be using the extensive use of data mining techniques. As per the statistics seven different levels of clones has been detected. Out of which some levels of detection are done manually thereby are known as simple clones [5] [6]. Other level needs some automation tools for detecting the clones. The levels that need concentration are as follows:

Level-1: Repeating groups of methods clones across different files.

Level-2: Repeating groups of file clones across different directories.

Level-3: Repeating groups of simple clones across different methods and files.

Level-4: Method clone sets and file clone sets.

This can be done by using Clustering and frequent item set mining without candidate generation with the help of FP-Growth algorithm [8]. The proposed algorithms and their performance results are given in the coming sections. And at last we just try to test the interface between the codes across different files and directories in order to identify the bugs and fix them by using model-based testing techniques.

The remaining section is organized as follows. In section 2, we give a procedure to identify the simple clones. In section 3, we briefly review about the FP-growth method and discuss about the algorithm and its usefulness in clone detection. In section 4, we introduce effective partition based clustering technique and its algorithm to search for clones at the directory level. Section 5, is dedicated to the results obtained by applying our technique to a particular project. And at last reference papers that have helped in guiding this paper are listed.

II. DETECTING SIMPLE CLONES

Grouping similar code fragments has been an easy task. These are done by pairing up the simple clone sets. As our methodology is based on the lexical analysis we use a tool called Repeated Tokens identifier to tokenize the given code into a string, from which a string based matching algorithm computes the simple clone sets. Our tool supports some languages like Java, PHP etc…

III. ORGANIZING THE CLONE DATA THROUGH FP-GROWTH ALGORITHM

Once the lower level clones are identified they need to be organized to make it compatible with the input format for the data mining technique that is applied on this data. We apply frequent item-set mining based on the market basket analysis

[10]. In this the input database consists of a list of transactions each one containing items bought together. Hence in our case a transaction corresponds to the simple clone sets that may be a file or a method. Although the previous work has been applied with the frequent item-set mining using the Apriori algorithm [7] it suffers from the following two drawbacks.

1. The methodology needs to generate huge number of candidate sets.
2. We need to repeatedly scan the database and check for a large set of candidates by pattern matching.

Hence in this scenario an interesting technique called frequent pattern growth (FP-Growth), which adopts a divide and conquer strategy is used. Level-3 and Level-4 clones can be detected using this technique. This method works as follows: First, it compresses the database representing frequent items into a frequent-pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases; each associated with one frequent item or "pattern fragment," and mines each such database separately.

For example let us consider a transactional database as shown in the figure-1,

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Fig 1. Transaction Database

In the first scan of the database we find out the 1-frequent itemset and their support count. Then the itemsets are listed in descending order of support count. Then we construct an FP-tree as follows. Initially create the root of the tree and label it with null. Then scan the database for the second time by creating the branches for each transaction. An item header for each node so that each item points to its occurrences in the tree through a chain of links.

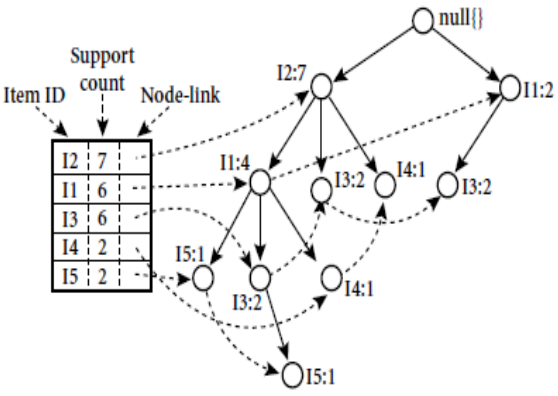

Fig 2. FP Tree

The mining procedure that is performed on the tree obtained above using the data can be explained with the help of the following algorithm.

Procedure FP_growth (Tree,α)

1) if Tree contains a single path P then
2) for each combination (denoted as β) of the nodes in path P
3) generate pattern βUα with support_count = minimum support count of nodes in β;
4) else for each a_i in the header of Tree {
5) generate pattern β= a_i Uα with support_count = a_i.support_count;
6) construct β's conditional pattern base and then β's conditional FP_tree $Tree_\beta$;
7) if $Tree_\beta \neq 0$ then
8) call FP_growth ($Tree_\beta$,β); }

But as the database grows in size it is sometimes better to first partition the database into a number of projected databases and then construct an FP-tree and mine it in each projected database.

IV. DETECTING FILE AND METHOD CLONES

Higher level clones i.e., clones existing at file and method level cannot be identified by FP-growth procedure because they are quite complicated. Hence Level-1 and Level-2 clones can be identified by using the clustering techniques. Clustering is a process of grouping the data objects into classes so that data objects within a class are highly similar to one another but dissimilar to data objects in other class based on attribute values describing these data objects. Although a number of techniques exist one of the best among them has been the Partitioning methodology. We have used the partitioning method by using K-means in this paper. In this procedure for a given database on 'n' objects and 'k' the number of clusters that are to be formed partitioning is done to organize the objects into 'k' partitions representing a cluster.

The following algorithm is applied to form clusters of data objects.

The k-means algorithm

(1) Arbitrarily choose 'k' objects from D as the initial cluster centers
(2) Repeat
(3) (Re)assign each object to the cluster to which the object is the most similar,
Based on the mean value of the objects in the cluster;
(4) Update the cluster means, i.e., calculate the mean of the objects for each cluster;
(5) Until no change;

This algorithm takes as input 'k' and partitions the data objects into 'k' classes. Most similar objects are grouped as a cluster and this process is repeated iteratively until it satisfies criteria given below.

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} |p - m_i|^2$$

Where, E is the sum of square-error for all objects in the database, 'p' is the point in space representing a given object, and 'mi' is the mean cluster of 'Ci'.

V. APPLICATION OF THE MODEL-BASED TESTING

In general many bugs may crop up during the application of these data mining techniques to the cloned data. And as a result the tool may not work in a correct manner. Hence to reduce the burden on the user we may like to apply a model-based test to the code. In this case a program model is used in some way to help with testing. Starting with the informal requirements specification, models with increasing levels of details are constructed.

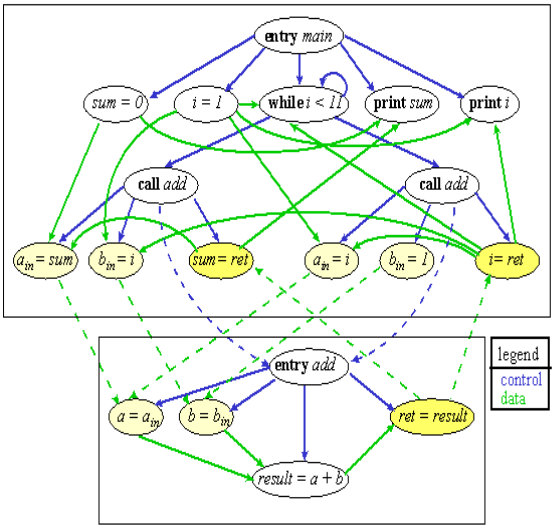

Fig 3. Model based Testing

As shown above application-specific code must be integrated into the framework [9] for which specialised framework classes are written and are executed for getting the clones information without any sort of bugs and errors. This part of the paper is still in progress so more details are not mentioned.

VI. IMPLEMENTATION

The architecture to implement the above mentioned techniques would appear as shown below in the fig.4.

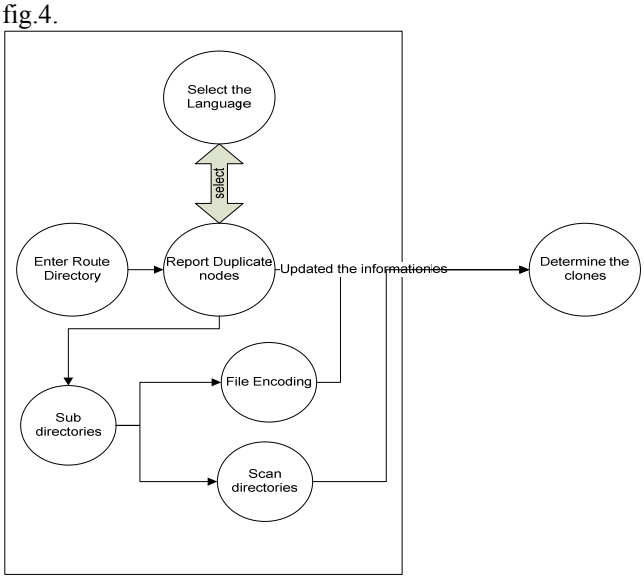

Fig 4. Clone Detection Architecture

In this architecture we would select a language of our interest and then enter the route directory of our choice of a particular project. A proper scanning of the sub-directories are performed using the above mentioned data mining techniques and apply proper encoding techniques and finally the updated information is reported to the user.

We have applied this process to a project named Tele-Medicine and we got the following result set. A sample file has been taken as showing the common lines of code.

```
public static void main(String a[]) {
    try{
      String add=reqCore();
System.out.println("aaaaaaaaaaaaaaaaaaaaaaa"+add);
    } catch(Exception e) { System.out.println(e); }
  }
public static String reqCore() throws Exception {
add=(String)obip.readObject();
          b=false;
          System.out.println("The Next Core Address
:"+add);
        }
         catch(Exception e) {
          e.printStackTrace();
        }
```

When the entire directory containing the project was taken and then scanned for detecting the clones the following data was obtained.

REPEATING GROUP OF METHOD CLONES ACROSS FILES

Minimum Cover	50%	90%
No. of groups	183	126
No. of file sets covered by groups	95	70
% of file sets covered	60%	40%
Files covered by groups	545	275
Min no of files in a group	1	1
Max no of files in a group	92	73
Avg. no of files in a group	25	30

VII. CONCLUSION

In this paper we emphasized on higher level cloning. The process is started by finding simple clones (that is, similar code fragments). Increasingly higher-level similarities are then found incrementally using data mining techniques of finding frequent closed itemsets, and clustering. We believe our technique is both scalable and useful. In this paper, we intend to extend our technique for testing the bugs at the interfaces of the clones. Implementing good visualizations for higher-level similarities is also an important part of our work. Currently, our detection and analysis of similarity patterns is based only on the physical location of clones. With more knowledge of the semantic associations between clones, we can better perform the system design recovery.

ACKNOWLEDGEMENTS

We would wish to thank our Head of the Department and the other faculty members who have helped us for this paper. I would also like to thank the selection committee who has reviewed this paper.

REFERENCES

[1] Kozaczynski, W., Ning, J. and Engberts, A. Program concept recognition and transformation. IEEE Transactions on Software Engineering, 18(12):1,065-1,075, December 1992.

[2] Baker, B. S. On finding duplication and near-duplication in large software systems. In Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE), pages 86-95, 1995.

[3] Ducasse, S, Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 109-118, 1999.

[4] Mayrand J., Leblanc C., Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 244-254, 1996.

[5] Krine, J. Identifying similar code with program dependence graphs. In Proceedings of the Eight Working Conference on Reverse Engineering (WCRE), pages 301-309. Stuttgart, Germany, October 2001.

[6] Koschke, R., Falke, R., and Frenzel, P. Clone Detection Using Abstract Syntax Suffix Trees. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE), pages 253-262, 2006.

[7] Kamiya, T., Kusumoto, S, and Inoue, K. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):654 – 670, July 2002.

[8] Rieger, M. Effective Clone Detection without Language Barriers. Ph.D. Thesis, University of Bern, 2005.

[9] Basit, H. A., Puglisi, S., Smyth, W., Turpin, A., and Jarzabek, S. Efficient token based clone detection with flexible tokenization. In Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE), pages 513-516, September 2007.

[10] Han, J., and Kamber, M., Data Mining: Concepts and Techniques, Morgan Kaufman Publishers, 2001.