2013 5th International Conference on Computational Intelligence and Communication Networks

Research on The Model of Legacy Software Reuse based on Code Clone Detection

Meng Fanqi
School of Information Engineering
Northeast Dianli University
Jilin, China
mengfanqi@mail.nedu.edu.cn

Kan Yunqi
School of Information Engineering
Northeast Dianli University
Jilin, China
Kan150@163.com

***Abstract*—The construction of large-scale engineering project usually relies on the development of many new software systems, whereas it would be very expensive and time-consuming if these new software systems are completely developed anew. In order to solve the problem, a model of legacy software reuse is proposed in this paper. The model is based on code clone detection. Firstly, code clone in source code of legacy software is detected by means of code clone detection tool. Secondly, abstract syntax trees of the functions which contain code clone are created. Thirdly, the degree of variation between the functions which contain the code clone belongs to the same clone set is calculated according to their abstract syntax trees, and then some functions whose similarities of abstract syntax trees are in the allowed range are combined. Finally, the combined functions and other frequently invoked functions are refactored into new functions or encapsulated into new classes, and all of these functions or classes can be reused as components in the development of new software systems. The test result shows that the reuse method based on this model can shrink the scope for searching the reusable component in legacy software systems, and thus improve the efficiency of legacy software reuse.**

Keywords- Legacy System; Code Clone; Refactoring

I. INTRODUCTION

One large-scale engineering project generally contains one or more subproject of software engineering. For instance, the construction of Smart Grid relies on the development of many new software [1], such as the software work for EMS (Energy Management System) and SCADA (Supervisory Control and Data Acquisition) etc. However, it must be very expensive and time-consuming if these new software are totally developed anew [2]. The study of the method to efficiently reuse legacy software in the same field may resolve this problem [3]. Although the definition of legacy software is not clear and definite, they usually have some common characteristics like large scale, complex structure and already running for a long time (more than 20 years). Since the development language of legacy software mostly is the third or early programming language (like ASM, COBOL or Turbo C etc.), and the development framework of legacy software has been outdated, the legacy software is hardly to be maintained and evolved. However legacy software may continue to contribute reusable code to new software which is similar in function. Most functions in legacy software are stability and credibility in processing the existing business, thus the method that reuses these functions in developing new software which can handle both existing business and emerging business has been adopted by many programmers [3].

Refactoring is a programming technique for optimizing the structure or pattern of an existing body of code by altering its internal nonfunctional attributes without changing its external behavior [4]. The software which comes through a series of successful refactoring can obtain some advantages, including reduced complexity and improved code readability to improve the maintainability of the source code, as well as a more clearly internal architecture to improve extensibility of the system function. The code refactoring of legacy software is one of basic methods to achieve efficient reuse. But the difficulty lies in how to identify the reusable code for the generation of component before the start of code refactoring. In this paper, we present a study of how to identify the reusable code from the legacy software by means of code clone detection tool. The remainder of the paper is organized as follows. The model of legacy software reuse is proposed in Section 2. And its kernel processes, code clone detection and components extraction are presented in Section 3. Section 4 analyzes the results obtained in a number of preliminary experiments and Section 5 outlines the conclusions and future work.

II. THE MODEL FOR CODE REUSE

The reuse of legacy software is a process to reengineering the old software system by component technology [3]. This process can be roughly divided into two steps: The first step is reverse engineering. Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction. It can also be seen as going backwards through the development cycle. Reverse engineering often involves taking computer program apart and analyzing its workings in detail to be used in maintenance, or to try to make a new program that does the same thing without using or simply duplicating (without understanding) the original. The second step is forward engineering. Forward engineering has the process similar to conventional development of software. It follows the flow: requirements analysis, outline design, detailed design, testing and modification. Fig. 1 shows the model built to reuse the legacy software based on code clone detection.

978-0-7695-5069-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CICN.2013.107

In this model, firstly, the change of requirement leads to the readjustment of architecture of legacy software system. This process can be divided into two stages, On the stage of requirement analysis, according to the change of requirement, Requirements Analysis Engineer increase new requirements or delete useless requirements based on the result of Requirement Analysis that comes from the reverse engineering of legacy software. On the stage of outline design, engineers readjust the architecture of legacy software to SOA (Service Oriented Architecture) for modernization [5]. Then, Architecture Readjustment needs new components which are compatible with the demand of new architecture and OOP (Object-Oriented Programming). Usually, three ways can be used to gain the needed components. The first one is to purchase from others; the second one is coding anew, the third one is reusing the code of legacy software. Just like the above mention that coding anew is time-consuming and purchasing may expensive, so the model prior adopts the method of identifying reusable code to extract component. Finally, new software will be developed with using component extracted from legacy software system.

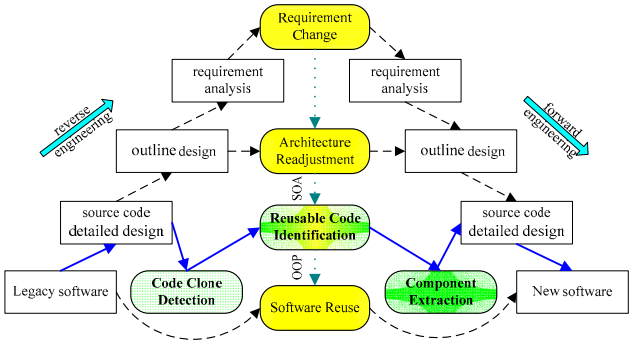

Figure 1. The model of legacy software reuse

III. COMPONENTS EXTRACTION

The reuse of legacy software is also known as Software Systems Modernization. Software Systems Modernization usually using SOA and Web Services update the framework and function for extending the lifetime of mission-critical legacy systems. Components play an important role in SOA [5]. Software engineers regard components as part of the starting platform for service-orientation. Actually, the reuse model uses component-based software engineering (CBSE) as the forward engineering method. Fig. 2 shows the refactoring process from component perspective.

A. Code Clone Detection

Since existing code has been tested well and may has less bugs then other code, programmer tends to reuse this kind of code in order to speed up the process of software development. When one code fragment was reused by passing without or with minor modifications, a clone pair was produced in the source code of the software, and this kind of practice for reuse existing code is so called code cloning. In the clone pair, the new code fragment copied from the original code is called a clone. However, In fact, which code is original and which code is clone is hardly identified in most cases. The above action is widespread, for instance, Baker in his paper announced that he has found between 13% - 20% of source code on large systems can be clone. And for COBOL an object-oriented system, the rate of duplicated code was even reach up to 50% [6] [7].

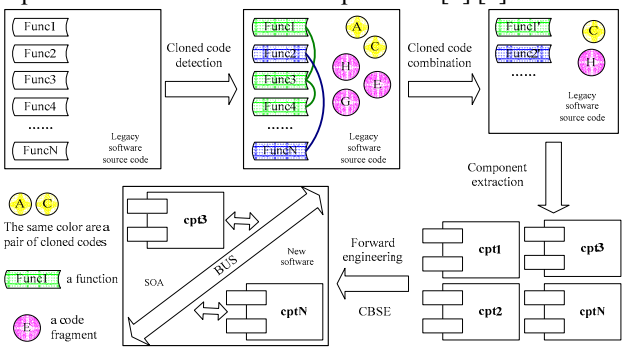

Figure 2. The process of component extraction

Although code clones may adversely affect the software systems quality, especially their maintainability and comprehensibility, the cloned code in legacy software are potentially most valuable code to be reused by refactoring it into new component. One piece of code is cloned more times, and then it generally has lesser bug and more useable value. So we should detect cloned code before identifying reusable component. In addition, code clone detection will compress the length of source code in legacy software, and reduce the work load in component extraction.

Many code clone detection methods and tools had been implemented. Such as text-based clone detection, token-based clone detection, abstract syntax tree (AST)-based clone detection, program dependency graph (PDG)-based clone detection, metric-based clone detection and many others. However, CCFinder is the one of the most famous and token-based code clone detection tool [8] [9] [10]. The work principle of CCFinder is followed: First, comments, blank lines and other uninteresting parts are removed from the source code in **preprocessing phase**. And then every thing except control word like “if” and “while” etc. in the source code is divided and replaced with special tokens in **transformation phase**. On the basis of tokenization, tokens of all source code are then concatenated into a single token sequence. After that, in **match detection phase**, a sub-string matching algorithm which bases on suffix-tree is then used to search the similar sub-sequences on the transformed token sequence. Once a similar sub-sequence pair is reported, the corresponding clone pair or clone class information is obtained with respect to the original source code. Finally, all clone pair locations are returned by line numbers and file location in **formatting phase** [10].

B. Abstract Syntax Trees Creation

Abstract syntax tree is a production generated after the lexical analysis and parsing of source code. Abstract syntax tree fully reflects the grammatical structure of the source code, and its leaves represent identifier or constant etc. Figure 3 shows an abstract syntax tree of a code segment.

Function is the basic unit in the third generation programming languages which are the main tools used in software development 20 years ago. Therefore the emphasis of reusing legacy software is functional refactoring for reuse. According to the reuse model, the functions include code clones which will be compared for calculating the similarity. Generally, the two functions with similar syntax structure are probably the same. So the abstract syntax tree of the functions includes code clone which should be built after code clone detection. Various tools for building abstract syntax tree can be downloaded easily from Internet, for example The GNU Compiler Collection and JavaCC.

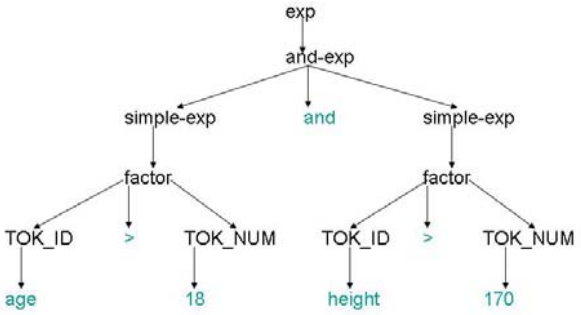

Figure 3. An example of abstract syntax tree

C. Differences Degree Calculation

The functions may be very different even though they have code clone belonging to the same clone set. The relationship between the function and the code clone probably has two cases. **In the first case**, the function is totally cloned (usually exists between different modules). **In the second case**, the function includes the code clone. Even though some large code clone may also include functions, but the larger code clone can finally be divided into the functions totally cloned and the small code clone pieces included by functions. In addition, the relationship between code clones which are detected by CCFinder also has three cases. **In case one**, the two code clones are the same. In **case two**, they are merely different in some identifiers' name. In **case three**, they may be minor different in variable types or syntax. All above cases happened because the detective method of CCFinder is token-based. Figure 4 shows an example of code clones, function 'foo1' and function 'foo3' have differences in some identifiers' names.

a) Totally cloned: If the function is totally cloned according to the detection result presented by CCFinder (it still has three cases discussed above), then we should calculate its difference degree with other related cloned functions by traversing abstract syntax tree twice. We directly compare the value of the node of the abstract syntax trees with the functions in the first traversing. If the result shows that the abstract syntax trees are the same, it means that the two functions are the same (note it as **Type A**). If not, we change all customer identifiers into $ when traversing the abstract syntax trees. If the result shows the same, the two functions are merely different in some identifiers' names (note it as **Type B**); else they are different in variable types or others (note it as **Type C**).

```
void foo1()
{int i,j,k;
double a[A][B], b[B][C], c[A][C];
    for (i=0; i<100; i++)
     {
      for(j=0; j<100; j++)
       for(k=0; k<100; k++)
          C[i][j] += a[i][k] * b[k][j];
     }
}
```

```
void foo3()
{int m,n,p;
double a[A][B], b[B][C], c[A][C];
    for (m=0; m<100; m++)
     {
      for(n=0; n<100; n++)
       for(p=0; p<100; p++)
          C[m][n] += a[m][p] * b[p][n];
     }
}
```

Figure 4. Example of code clones

b) Partly cloned: If the function is partly cloned, it means that the function includes code clone in its body. We traverse the abstract syntax tree of the function with changing customer identifiers into a token. Those functions which include the code clone belonging to the same clone set will be compared with traversing results. We adopt Levenshtein Distance[11] (or Edit Distance) for the compare method. The algorithm of the calculation of Levenshtein Distance between the two string fp1 and fp2 is shown as Figure5. The different degree between two functions can be gotten via calculating the expression: **DD= (matrix (len1, len2)/max (len1, len2))*100%**. The bigger the value of DD is, the more different the two functions are. We can use a threshold value to decide whether the functions are similar. If the value of DD is below the threshold value, note the two functions as **Type D**, else noted them as **Type E**.

1: $len1 \leftarrow strlen(fp1)$
2: $len2 \leftarrow strlen(fp2)$
3: $initialize\_two\_dimensional\_matrix(matrix, len1, len2)$
4: **for** $i = 0 \rightarrow len1$ **do**
5: **for** $j = 0 \rightarrow len2$ **do**
6: **if** $fp1[i] = fp[j]$ **then**
7: $cost = 0$
8: **else**
9: $cost = 1$
10: **end if**
11: $matrix[i, j] = min(matrix[i-1, j]+1, matrix[i, j-1]+1, matrix[i-1, j-1]+cost)$
12: **end for**
13: **end for**
14: **return** $matrix(len1, len2)$

Figure 5. The algorithm of Levenshtein Distance calculation [11]

D. *Cloned functions combination*

A cloned function is a function with code clone. The combination of cloned functions can be divided into five cases according to the types of cloned function.

- In **Case 1**, the functions which will be combined are **Type A**. In this case, all of the functions are the same, so we select one of them and add it into function base.
- In **Case 2**, the functions which will be combined are **Type B**. In this case, all of the functions are very similar except individual identifier's name, so we select the shortest one for saving space and add it into function base.
- In **Case 3**, the functions which will be combined are **Type C**. In this case, the differences between the functions are in types of variable or in other aspects, so we select the longest one for retaining enough information and add it into function base.
- In **Case 4**, the functions which will be combined are **Type D**. In this case, even though the difference degree is lower than a preset threshold value, the functions are more different with each other than Type A, Type B and Type C. so we should flexibly adopt various existent refactoring method to combine the functions. The combined function will be added into function base.
- In **Case 5**, the functions are **Type E**. Since the similarity is too low, the functions are not recommended to combine. All of the functions are respectively added into function base.

All of the functions in the function base have a value which denotes invoked times in legacy software. The value is an important reference for component extraction. In Case 1 to Case 4, the invoked time of a combined function is the sum of invoked times of all related cloned functions.

E. *Component extraction*

In this paper, we consider a component as a module that encapsulates a series of application area related functions (or data). Programmers can use these functions which have been stored in function base as various forms. For example, some functions can be encapsulated into a new class as function members by tiny modification, or assembling some functions to generate a DLL files. In addition, several tools have been used in extracting components, such as CodeMiner, CARE (Computer-Aided Reuse Engineering) and PATRicia (Program Analysis Tool for Reuse).

IV. ANALYSIS AND RESULT

We did some preliminary experiments about cloned code detection which is the basic work in the model. We selected mysql-5.1.45 and postgresql-8.1.17 as our experimental subjects. The two applications were both written in C or C++ and they are both applications in database. We used CCFinder to detect code clone hidden in the two applications. Before the detection, the value of Minimum TKS was set at 30. These values were more suitable for two reasons: firstly, the codes will be no much reuse value if its length is too short; secondly, the code clones usually do not have a length longer than 50 TKS. 148 clone sets in mysql, 56 clone sets in postgresql and 3 clone sets in both applications were detected and their metric results are shown as Table 1. The meaning of the Names in the tables can be found at http://www.ccfinder.net/doc/10.2/en /tutorial-gemx.html

TABLE I. CLONE SET METRICS

Name	Min.	Max.	Average
LEN	86	3377	456.517
POP	2	6	2.23188
NIF	1	6	1.75845
RAD	0	6	1.48309
RNR	0.114	1.000	0.755733
TKS	30	53	34.2319
LOOP	0	11	1.95169
COND	0	87	9.19324
McCabe	2	87	11.1449

In the 207 clone sets, all clone types according to the type define in this paper were found. We also found that **Type A** clone usually exist between the files which have the same or similar name. Figure 6 shows an example of Type A clone. The two functions "mem_heap_strcat" are identical and both in file "memOmem.c", but the left one is in "innobase" folder and the right one is in "innodb_plugin" folder.

1003 H:\TestObj\mysql-5.1.45\storage\innobase\mem\memOmem.c

```
155 char*
156 mem_heap_strcat(
157 /*============*/
158                 /* out, own: the result */
159     mem_heap_t* heap,   /* in: memory heap where string
160     const char*   s1, /* in: string 1 */
161     const char*   s2) /* in: string 2 */
162 {
163     char*  s;
164     ulint   s1_len = strlen(s1);
165     ulint   s2_len = strlen(s2);
166
167     s = mem_heap_alloc(heap, s1_len + s2_len + 1);
168
169     memcpy(s, s1, s1_len);
170     memcpy(s + s1_len, s2, s2_len);
171
172     s[s1_len + s2_len] = '\0';
173
174     return(s);
175 }
```

1208 H:\TestObj\mysql-5.1.45\storage\innodb_plugin\mem\memOmem.c

```
132 char*
133 mem_heap_strcat(
134 /*============*/
135     mem_heap_t* heap,   /*!< in: memory heap where stri
136     const char*   s1, /*!< in: string 1 */
137     const char*   s2) /*!< in: string 2 */
138 {
139     char*  s;
140     ulint   s1_len = strlen(s1);
141     ulint   s2_len = strlen(s2);
142
143     s = mem_heap_alloc(heap, s1_len + s2_len + 1);
144
145     memcpy(s, s1, s1_len);
146     memcpy(s + s1_len, s2, s2_len);
147
148     s[s1_len + s2_len] = '\0';
149
150     return(s);
151 }
152
```

Figure 6. An example of Type A function clone from the fileshave the same name but in different folders

When the Minimum TKS was reset at 15 and the two applications were redetected, we even found more reusable code from the two applications. Figure 7 shows an example of Type B cloned function exists in both applications. The two functions have the same name 'yy_flush_buffer', but the left one is in mysql-5.1.45 and the right on is in postgresql-8.1.17. They are different in the name of identifier 'yy_current_buffer' at line 2494 and line 2384. They are **Type B** clone and have the same size, so according to the model of legacy software reuse, we can select one of them random for component extraction. Figure 8 shows the function 'yy_load_buffer_state' which is called by the function 'yy_current_buffer' showed in figure7. In fact, Figure 8 is an example of **Type C** .

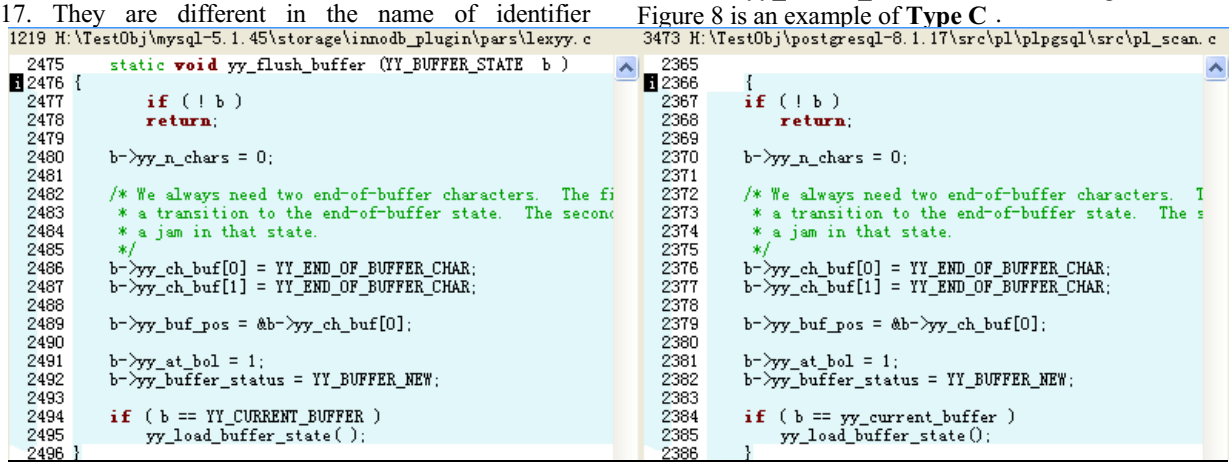

Figure 7. An example of Type B function clone from different files of different applications

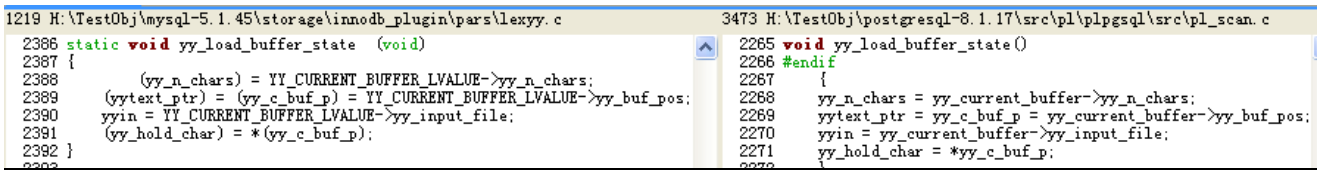

Figure 8. An example of Type C function clone from different files of different applications

V. CONCLUSIONS

Legacy software may help the construction of large-scale engineering project in efficiencies and costs, but it depends on whether the legacy software can be reused. In order to reuse the legacy software efficiently, we proposed a reuse model based on code clone detection. By detecting code clone, we can firstly reduce the candidates for component extraction, thereby lower the complexity; secondly, the remained cloned functions after function combination are more valuable for components generation, thus enhance the reliability of the refactoring. However, the result shows that the valuable cloned functions are not too much in legacy software, so the method of this model should be used as a subsidiary method in refactoring large-scale legacy software.

REFERENCES

[1] Singhal, A.; Saxena, R. P., "Software models for Smart Grid," *Software Engineering for the Smart Grid (SE4SG), 2012 International Workshop on* , vol., no., pp.42,45, 3-3 June 2012

[2] Valerdi, R., "Heuristics for Systems Engineering Cost Estimation," *Systems Journal, IEEE* , vol.5, no.1, pp.91,98, March 2011

[3] Gan Wang; Valerdi, R.; Fortune, J., "Reuse in Systems Engineering," *Systems Journal, IEEE* , vol.4, no.3, pp.376,384, Sept. 2010

[4] Soares, G.; Gheyi, R.; Massoni, T., "Automated Behavioral Testing of Refactoring Engines," *Software Engineering, IEEE Transactions on* , vol.39, no.2, pp.147,162, Feb. 2013

[5] Jaejoon Lee; Kotonya, G.; Robinson, D., "Engineering Service-Based Dynamic Software Product Lines," *Computer* , vol.45, no.10, pp.49,55, Oct. 2012

[6] Monden, A.; Okahara, S.; Manabe, Y.; Matsumoto, K.-i., "Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations," *Software, IEEE* , vol.28, no.2, pp.42,47, March-April 2011

[7] Hoan Anh Nguyen; Tung Thanh Nguyen; Pham, N.H.; Al-Kofahi, J.; Nguyen, T.N., "Clone Management for Evolving Software," *Software Engineering, IEEE Transactions on* , vol.38, no.5, pp.1008,1026, Sept.-Oct. 2012

[8] Sarkar, M., Mondal, T., Roy, S., Mukherjee, N.," Resource requirement prediction using clone detection technique". *Future Generation Computer Systems*, vol.29,no.4,pp.936–952.2013

[9] Dongxiang Cai, Miryung Kim, "An Empirical Study of Long-Lived Code Clones", *International Conference on Fundamental Approaches to Software Engineering*, 2011, pp. 432–446.

[10] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", *Transactions on Software Engineering*, Vol. 28, no.7, 2002,pp. 654- 670.

[11] Wu Zhou, Yajin Zhou, Xuxian Jiang, "Detecting Repackaged Smartphone Applications inThird-Party Android Marketplaces", *In Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*,2012,pp.120-130.