IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 3, March 2017

Critical Review of the Bunch: A Well-Known Tool for the Recovery and Maintenance of Software System Structures

Mahjoubeh Tajgardan[1], Habib Izadkhah[2]

Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran[1, 2]

Abstract: Lack of up-to-date software documentation hinders the software evolution and maintenance processes, as simply the software structure and code could be easily misunderstood and not comprehended. One approach to overcome such problems is to extract and reconstruct the software structure from the available source code so that it can be assessed against the required changes. Such approach is known as Software clustering. Bunch is a well-known tool for structure reconstructing and maintaining of software systems. As the maintenance of software costs a lot, Bunch is usually employed to reconstruct the structure of software and consequently making required changes. The aim of this paper is to investigate the strengths and weaknesses of the Bunch; if problems and proposals offered in this paper are studied and solved, this tool will be turned to be a useful tool for structure recovering and maintenance of a software system.

Keywords: Software System; Software Structure; Bunch; Software maintenance.

I. INTRODUCTION

Many of businesses, governments, and social institutions are software-related work to do. When the requirements of these institutions change, the software must to adapt itself to it. Changing large and complex software systems, which can be thousands or even millions of lines of code, can be quite difficult. During software maintenance and evolution processes, the actual software architecture could deviate from the originally documented architecture in order to fulfill the changing business requirements. Such architecture changes are not necessarily well documented, and in some extreme cases are not documented at all, as in legacy software systems. This of course leads to software miscomprehension, which hinders the software future evolution and maintenance processes. Muller, et al., (1995) shows that 50–90% of software evolution work focuses on program comprehension. Therefore, program comprehension plays an important role in software development [1]. Comprehending the available program is possible through preparing a design attribute such as software architecture [2].

Bunch proposed by Mitchell in his Ph. D thesis [3, 4]. Bunch is a clustering tool that provides developers higher-level structural information about the numerous software subsystems (or modules, i.e. a number of interrelated classes), their interfaces, and their interconnections for better understanding of software structure to maintain and apply changes. In this tool, subsystems can also be organized hierarchically, allowing developers to study the structure of a system at various levels of detail by navigating through the hierarchy. The main reason of choosing the Bunch is that this tool is widely used in software industry and is applied as base software in order to compare other algorithms; also to date it has been cited over 300 times by other researchers.

II. OVERVIEW OF THE BUNCH

Fig. 1 shows the structure of the Bunch toolset.

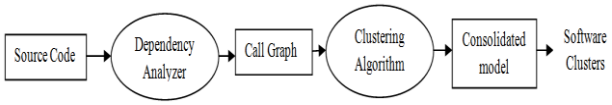

Fig. 1 Structure of Bunch tool

As indicated in Fig. 1, the extraction steps through Bunch are as follows: first, the source code is analyzed and then call graph is generated from it. Bunch uses Acacia algorithm to generate Class Dependency Graph (CDG) for C++ programming language and Chava for JAVA programming language. Second, after the call graph generation, it clustered using genetic algorithm to find an appropriate structure. In Bunch, the aim of clustering is minimizing connections between the classes of two different modules (called coupling), as well as maximizing connections between the classes of the same module (called cohesion). Generally, low coupling and high cohesion are considered as characteristics of well-designed software systems [5]. For this purpose, there exist two functions named BasicMQ and TurboMQ for computing quality of obtained clustering in Bunch. If A_i is internal connection level for a module and E_{ij} represents connection level between two modules "i" and "j", then having a program graph divided to "k" modules, BasicMQ is defined as follows:

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 3, March 2017

$$\text{basicMQ} = \frac{1}{k}\sum A_i - \frac{1}{\frac{k(k-1)}{2}}\sum E_{ij} \quad (1)$$

Definitely, the clustering that increases the value of this function would enjoy higher quality. BasicMQ criterion is bounded in the interval between -1 (no cohesion in subsystems) and 1 (no coupling between subsystems). If CDG has |V| classes and |E| dependencies, the computational complexity of BasicMQ will be: $O(|V|^2 \times |E|)$. Efficiency of this criterion is low because of having high operational costs limiting its application to small systems (systems having less than 75 modules).

TurboMQ criterion has been designed to overcome limitations of BasicMQ. If the internal edges of module and edges between two modules are respectively represented by μ_i and $\varepsilon_{i,j}$, TurboMQ value will be then computed as follows:

$$TurboMQ = \sum_{i=1}^{k} CF_i \quad (2)$$

$$MF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{j=1}^{k}(\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases} \quad (3)$$

Third, Due to the heuristic nature of Bunch, it may produce results with the same quality but different clustering for different runs on a given graph. In different clustering process, is observed regular displacement of several classes between different modules while other classes displace less. Bunch represents software structure more precisely upon results of numerous clustering by an algorithm or several different algorithms and derivation of common patterns of them. The Bunch uses Precision/Recall for this purpose. This criterion, measures the similarity between two clustering based on co-modules pairs in clustering. Supposing we want to compare two A and B clustering, precision will be the percent of co-module pairs in A that also are co-module in B, and recall is the percent of co-module pairs in B that also are co-module in A. In these criteria, maximum Precision is the prior one. If this value was the same for two modules, modularization should include less Recall. Forth, Bunch can show obtained structure as a graph or in UML.

III.CALL DEPENDENCY GRAPH ALGORITHM IN BUNCH

The input of Bunch is the CDG generated from the source code. As mentioned in Section 2, Bunch uses Acacia and Chava to generate CDG. Main problems of these algorithms are that they do not consider the implicit calls in designing. These algorithms could not construct CDG precisely when a source code includes implicit calls. Fig. 2 shows pseudo code, which does not include implicit call while Fig. 3 shows implicit call in a pseudo code. In Fig. 2, declared type for variable "a" is class "A" and "a" instantiated of class "A". Thus call destination a.method1( ) is considered class "A". While in Fig. 3, declared type of "a" is class "A but "a" instantiated of class "B". Thus, call destination a.method1( ) should be considered class "B" not class "A". In such cases, existing algorithms consider both classes as call destination. In fact, dependence graph is produced pessimistic. This kind of call is called implicit call. Consider the code in Fig. 4.

```
Class A {
        Public void method1 ( ){ print ("This is A");}
         }
Class B extends A {
        Public void method1 ( ){ print ("This is B");}
        }
Class C extends A {
        Public void method1 ( ){ print ("This is C");}
        }
Class Main {
Public void method2 ( ){
    A a;
    a=new A ();
    a.method1 ();
                }
}
```

Fig. 2 A pseudocode without implicit call

```
Class A {
        Public void method1 ( ){ print ("This is A");}
         }
Class B extends A {
        Public void method1 ( ){ print ("This is B");}
        }
Class C extends A {
        Public void method1 ( ){ print ("This is C");}
        }
Class Main {
Public void method2 ( ){
    A a;
    a=new B ();
    a.method1 ();          // implicitly calls
                }
}
```

Fig. 3 A pseudocode includes an implicit call

```
class A {…. }
class B extends A {…. }
class C{
        public void  m1(){
                A b=new B();
                b. m();
                }
        }//class c
class main_Class{
        public Static void main(){
                A a=new A();
                a. m();
                C c=new C();
                c. m1();
```

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 3, March 2017

```
        }//main
    }//main_class
```

Fig. 4 Example of source code

Fig. 5 shows CDG generated for Fig. 4 by Chava algorithm. This graph constructed so pessimistically.

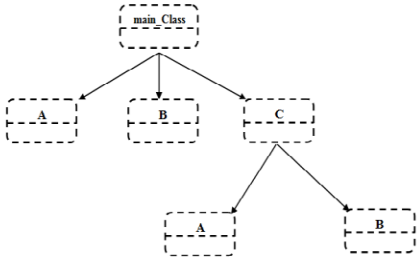

Fig. 5 CDG generated for Fig. 4 by chava algorithm

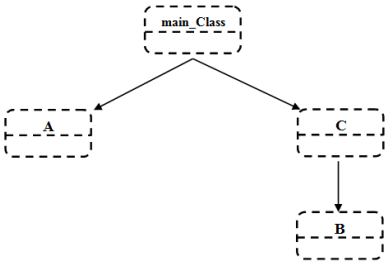

Fig. 6 Appropriate CDG for Fig. 4

The appropriate CDG for Fig. 4 has shown in Fig. 6. Algorithms used for constructing act pessimistically. These algorithms construct the CDG conservatively and do not eliminate any probable call from the graph. As a result, the obtained CDG will have so many edges and will have negative impact on the clustering result in Bunch. The reason for this is that the aim of clustering is to make sub-systems with maximum cohesion and minimum coupling and whatever the number of edges in a graph is much more, making sub-systems that can be organized as cohesive modules that are loosely inter-connected will be more complex and also will slow down finding proper architecture.

IV. CLUSTERING ALGORITHM IN BUNCH

The general problem of graph partitioning (of which software clustering is a special case) is NP-hard [4]. For this reason, Bunch uses search-based technique for clustering. Since searching the complete state space turns the situation into a NP-Hard problem, in the Bunch genetic algorithm is used for finding the optimal or near optimal software structure during a reasonable time. Search operation is carried out using criteria of maximal cohesion and minimal coupling of clusters.

Efficient behavior of a genetic algorithm depends on proper design of encoding. Each node in the CDG has a unique numerical identifier assigned to it (e.g., node N_1 is assigned the unique identifier 1, node N_2 is assigned the unique identifier 2, and so on). These unique identifiers define which position in the encoded string is used to define that node's cluster. Suppose S: = 2 2 4 4 1; therefore, the first character in S, i.e., 2, indicates that the first node (N_1) is contained in the cluster labeled 2. Likewise, the second node (N_2) is also contained in the cluster labeled 2, and so on. Formally, an encoding on a string S is defined as:

$$S = s_1\ s_2\ s_3\ s_4\ \ldots\ s_N \quad (4)$$

Where N is the number of nodes in the CDG and s_i identifies the cluster that contains the i^{th} node of the graph.

One of the important issues related to the Bunch algorithm is largeness of search space due to presence of some repetitive answers, i.e. although some generated codes have apparently different representations, but in reality, they represent the same partition. For example, though two chromosomes S_1= 2 2 4 4 1 and S_2=1 1 5 5 3 have different appearances but they are actually representative of the same partition. Because, in both, there are three modules so that nodes of N_1 and N_2 are in same module, nodes of N_3 and N_4 are in same module and node node N_5, located in distinct module. Search space in Bunch algorithm is n^n; this large search space decelerates speed of this algorithm to find appropriate structure. The state space of n^n is the worst state for a problem. State space search in this state is impossible in a rational time. Such state space would cause doubt in finding a good structure for software by Bunch. We believe that we can reduce it using an appropriate representation (i.e. encoding). For example, if we use the following representation (Fig. 7), the state space will be reduced to n!. This significant reduction may have a significant effect on improvement of the quality of achieved structure.

Here, m^{th} cell of array (encoding) represents the class number "m" of the call graph. Its content includes number of another node of graph like "p" and if "p" is equal or greater than "m", then "m" is placed in a new module; otherwise "m" belongs to the same module as "p".

Class Number (m)	1	2	3	4	5	6
p	1	5	6	3	2	4

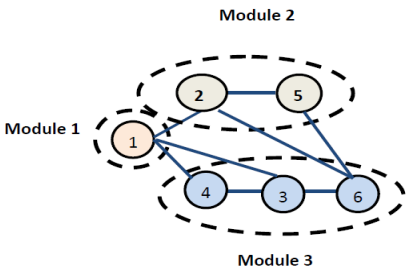

Fig. 7 Chromosome structure and modules obtained from encoding

A. Computing the Quality of module

As mentioned is Section 2, Bunch uses BasicMQ and TurboMQ functions to calculate the quality of the obtained

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 3, March 2017

modules. To calculate this function should be calculated internal and external communications of a module. In general, there are three types of relationships between classes as follow:

- **Aggregation:** are of the form class-attribute as a class D is the field of class M.
- **Class-method:** in this case, class D is the type of a parameter of method m_c of a class C, or if a class D is the return type of method m_c.
- **Method-method:** in this case, method m_d of a class D directly invokes a method m_c of a class C, or a method m_d receives via parameter a pointer to m_c thereby invoking m_c indirectly.

In TurboMq, is considered same the influence of all kinds of relations among modules. But, we think that Aggregation is more important than class-method and the significance of this relation is higher comparing with method-method. Hence, it is recommended to consider their impact on clustering different. In other word, beside their number, their weight should be taken into account according to the rate of value as well. The reason is that If have two classes, Aggregation relationship, they should be clustered in same module. Thus, we suggest the MF_m can be calculated as follow.

$$MF_m = \frac{2(\sum_{i=1}^{3} w_i \mid C_i \mid)}{2(\sum_{i=1}^{3} w_i \mid C_i \mid) + \sum_{i=1}^{k} (\sum_{p=1}^{3} w_p \mid (C_{i,j})_p \mid + \sum_{p=1}^{3} w_p \mid (C_{j,i})_p \mid)} \quad (5)$$

C_1= class-attribute and $|C_1|$= number of class-attributes in the source code, w_1= weight of C_1
C_2= class-method and $|C_2|$= number of class-methods in the source code, w_2= weight of C_2
C_3= method-method and $|C_3|$= number of method-methods in the source code, w_3= weight of C_3
$|C_{i,j}|$ is the representative of call numbers from module i to module j and $|C_{j,i}|$ is the representative of call numbers from module j to module i.

V. CONSOLIDATED MODEL IN BUNCH

As mentioned in Section 2, Bunch uses Precision/Recall to improve the quality of the obtained modulus. The problem here is that criterion value is susceptible to size and number of modules and sometimes would lead to less precise result. Suppose that the main structure of a system is as Fig. 8(a). Also suppose Fig. 8(b), 8(c), and 8(d) are the obtained structure from a clustering algorithm. Precision/Recall value of each structure has been written beneath it. The obtained Precision/Recall value for Fig. 8(c), shows that it is the most similar structure to Fig. 8(a) structure, But unlike results of Precision/Recall, Fig. 8(b) is the closest structure to the main structure. Because, the only difference between Fig. 8(b) and Fig. 8(a) is that class c_4, which has been moved to cluster B_2. We propose to overcome this problem in Bunch; it uses harmonic mean of Precision/Recall. Harmonic mean of Precision/Recall is calculated as relation 6.

$$F_M = 2\frac{Precision * Recall}{Precision + Recall} \quad (6)$$

Now, if F_m is calculated for each clustering in Fig. 8, it is observed that 8(b) has the maximum F_m value and the closest structure to the main structure, too.

VI. OTHER CASES

There is an idea that by adding the following cases to the Bunch, it would be more efficient:

1- We believe that user should interfere in clustering process and lead it to his/her interests, some of which includes the following cases:

1) Setting upper bound and lower bound on the number of modules,
2) Two or more specific classes sharing the same module,
3) Two or more specific classes not appearing in the same module,
4) Limiting the sizes of the modules,
5) Bounds on the number of classes in each module.

Above-mentioned cases can be added as constraint to Bunch objective function. This is achieved by using a general penalty function. Penalty function can be defined as normal distribution function.

2- Program clustering plays an important role in automatic distribution of sequential code. We believe that maximum cohesion and minimum coupling criteria are not appropriate for the evaluation of the quality of the architecture of distributed codes. This kind of architecture is aimed at achieving the shortest execution time by maximizing the degree of concurrency in execution of the distributed code. We think to achieve the maximum possible speed up, should changed the Bunch objective function. Bunch objective function shall maximize asynchrony calls and minimize synchrony calls.
Clustering of source code will be in a way to improve security of software system instead of obtaining maximum cohesion and minimum coupling, which is the objective of classic architecture. Istehad Chowdhury, et al. [6] in their research on Mozilla Firefox empirically shows that a significant relationship exists between coupling and security. In Our opinion if we derive five coupling dependencies, namely data, control, content, common and stamp, from source code, then dependency of these criteria and security can be defined as a mathematical relation. This relation could be used as objective function for clustering based on security.

VII. CONCLUSION

In this paper, we evaluated the Bunch, a well-known tool for recovering the structure of a software system. It consists of three steps: the call graph generation, genetic algorithm for clustering and consolidated model. The main

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 3, March 2017

problem of this tool in generation of call graph is that it cannot identify the implicit calls from the program. So, call graph is produced pessimistically which has reverse impact on clustering results. On the other hand, state space in Bunch to find software structure is n^n which has a considerable negative influence on finding an optimum structure for software system. To calculate the quality of clusters, Bunch uses BasicMQ and TurboMQ by which there would be no difference between Aggregation, Class-method and method-methods, while their influences on software structure are different. Bunch can achieve a good structure using Precision/Recall criterion. This criterion is sensitive to the number of clusters and in some cases it may lead to less accurate results.

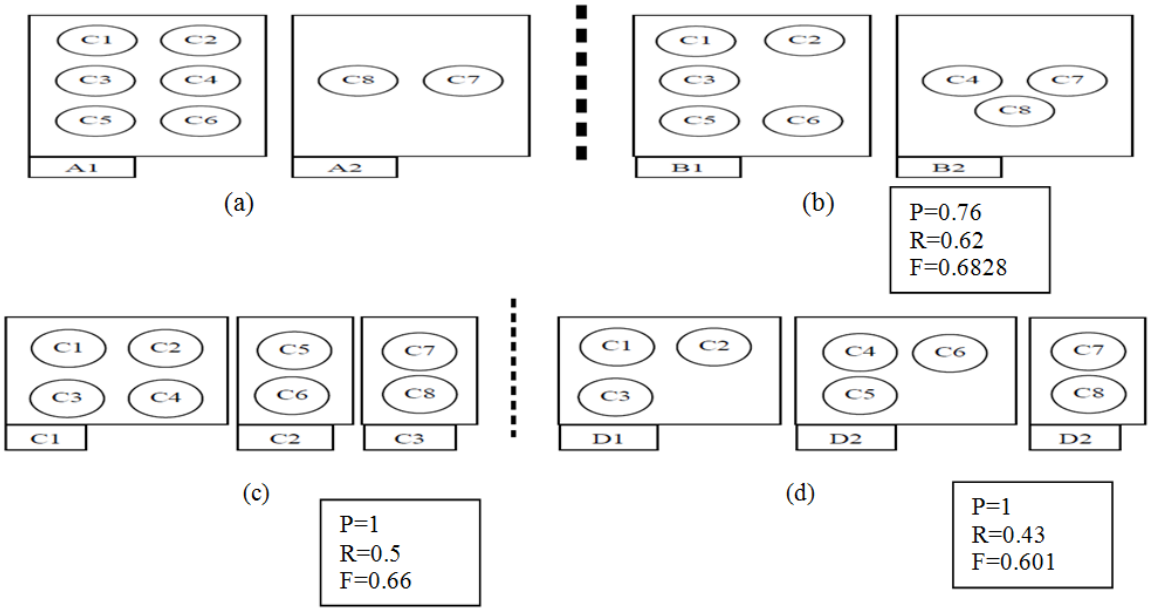

Fig. 8 Some structures of a system

REFERENCES

[1] Z. Qifeng, Q. Dehong, T. Qubo, S. Lei, Object-Oriented Software Architecture Recovery Using a New Hybrid Clustering Algorithm. 2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2010), Pages 2546-2550.

[2] S. Ducasse S, and D. Pollet, Software Architecture Reconstruction: A Process - Oriented Taxonomy. IEEE Transactions on Software Engineering, Vol. 35, No. 4, 2009, Pages 573-591.

[3] S. Mancoridis, B.S. Mitchell, Y. Chen, E.R. Gansner, Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures, ICSM '99 Proceedings of the IEEE International Conference on Software Maintenance, 1999.

[4] B.S. Mitchell, A Heuristic Search Approach to Solving the Software Clustering Problem. Ph.D Thesis, Drexel University, Philadelphia, 2002.

[5] R.S. Pressman, Software Engineering: A Practitioner's Approach. 7th ed. McGraw-Hill, Inc, 2010.

[6] I. Chowdhury, and M. Zulkernine, Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities. Journal of Systems Architecture 57, Pages 294-313, 2011.