Recover and Optimize Software Architecture Based on Source Code and Directory Hierarchies

Tong Wang, Yelian Zhang, Xufang Gong, Bixin Li
School of Computer Science and Engineering
Southeast University, Nanjing, China

***Abstract*—Software architecture helps developers understand and maintain software, so how to obtain accurate architecture is critical. The architecture recovery technique is a widely used method for obtaining architecture. In order to improve the accuracy and efficiency of the architecture recovery technique, we propose a method for recovering and optimizing software architecture based on source code and directory hierarchies. Our method consists of three steps: first, we extract architecture-related information from source code and directory hierarchies and construct a file dependency graph; then, we preprocess and cluster code elements to construct a preliminary architecture; finally, we optimize architecture by clustering code elements based on the structural similarity and renaming components. We perform our method on four representative open source projects and compare our method with representative architecture recovery techniques. The experimental results show that the architectures recovered by our method has higher accuracy with higher efficiency than the compared architecture recovery techniques.**

***Index Terms*—architecture recovery; clustering; architecture optimization**

I. Introduction

Software architecture provides a high-level abstraction view to developers, so it is useful for understanding software to make evolution plans, reduce development costs, improve software performance [1] and improve software quality [2], [3], so how to obtain accurate software architecture is a hotspot issue.

The automated recovery technique mainly includes two phases, information extraction and information-based recovery. In the information extraction phase, the data sources commonly include source code and directory hierarchies [4]. The directory hierarchy is set by developers, so it reflects the logical relations between files [5]. These data sources provide valid information for software architecture recovery. However, most architecture recovery techniques usually only use one of them to recover architecture resulting in the low accuracy. In the information-based recovery phase, there are many types of recovery algorithms. However, there is a huge amount of code elements in large scale programs, so how to improve efficiency is also an important problem.

In summary, the current automated architecture recovery technique has the following problems. a) The data source is single. When information is missing, the accuracy would be reduced. b) In a large scale program, the number of code elements is huge, so it will take a long time to recover architecture.

In order to solve the above problems, we propose the method for architecture recovery and optimization based on source code and directory hierarchies (ARO). The main contributions of this paper are mainly reflected in the following three aspects.

- The architecture-related information is extracted from source code and directory hierarchy to improve accuracy. The information of source code reflects the specific implementation of the architecture. The information of directory hierarchies is related to developers' design.
- The file dependency graph is preprocessed to improve efficiency. The file dependency graph is preprocessed to reduce the number of clustered code elements for reducing the time consumed, then we perform K-center-hierarchy algorithm to cluster code elements around the core code.
- The architecture is optimized. We optimize architecture by clustering based on the structural similarity and renaming components.

II. Related work

The automated recovery technique mainly includes two phases, information extraction, and information-based recovery.

Mainstream data sources contain the following types. a) Source code. Text analysis or feature localization of the source code of the program by constructing a platform similar to lexical analysis and grammar analysis, and the use of information retrieval technology to discover associations between documents. b) Documents. Documents related to software are collected, such as software design documents, UML diagrams, code comments, user manuals, etc., to establish the concept [6]. c) Directory hierarchy. Some methods consider the information to identify components, that is, it is a supplement to the information obtained by other methods and improves the accuracy of partitioning components [4].

Mainstream methods of information-based recovery contain the following types. a) Domain knowledge. Domain-based methods are performed in a top-down process or a bottom-up process. In the top-down process, the documents related to architecture materials are used to recover architecture. In the bottom-up process, according to the comments, the declaration of variable names, etc., using domain knowledge to recover

B. Li is the corresponding author. E-mail: bx.li@seu.edu.cn
DOI reference number:10.18293/SEKE2019-045

architecture [7]. b) Clustering. This type of method uses mathematical methods to study and process the classification [8]. c) Machine Learning. The information of code elements is trained to recover architecture. Machine learning-based methods are generally not used alone, but as a supplement to clustering algorithms to improve the accuracy of clustering. d) Pattern Matching. The recovery process is modeled as a mapping between the high-level abstraction and the code elements. It is a semi-automated technique that requires manual participation, and graph matching also requires a lot of computer resources and time [9].

According to the above analysis, we find that only the clustering algorithm and the machine learning method do not require additional manual intervention. However, it is difficult to obtain the training set [10]. Therefore, the clustering algorithm is more widely used in the automatic recovery architecture technique.

III. Our method

Our method contains the following steps: extracting information, preprocessing and clustering code elements, and optimizing architecture.

A. *Extracting information*

The information of source code is extracted by automatically analyzing source code, such as the dependency information between code elements and the basic information of code elements. The process of extracting information from source code contains two steps. a) Construct an abstract syntax tree. The source code is converted into an abstract syntax tree which is an intermediate representation. b) Analyze the abstract syntax tree. By analyzing the abstract syntax tree, the information between the code elements is extracted. A code element is a unit of code, such as a file, a package, and so on. The dependency between code elements can be divided into multiple types, such as the reference dependency between files, the generalization dependency between classes, and so on. Then we integrate dependency information to construct the file dependency graph.

The information of directory hierarchies reflects the logical dependencies between files, and it is mainly used in the following two aspects: a) Aggregating components. In source code, some files do not have dependencies with other files, such as test case files. These files are presented as an independent component in architecture. A large number of independent components have effects on the understandability, so we cluster independent files into an independent component. b) Adjusting components. The relations between files of the same directory do not strictly comply with the high cohesion principle and the low coupling principle, so we adjust components based on the dependencies between directories.

B. *Preprocessing and clustering code elements*

Before we perform the clustering algorithm, we preprocess the file dependency graph to reduce the number of files, resulting in reducing the time consumed.

In the preprocessing process, independent files are clustered as early as possible, which is beneficial to reduce the number of nodes in the dependency graph and to comply with the high cohesion principle and the low coupling principle. Then, we preprocess the file dependency graph by clustering code elements which are related to the types of strong dependency and the structure of strong dependency. The dependency type between code elements indicates the degree of the closeness between them. The common types of strong dependency between code elements contain the following types: the generalization dependency, the implementation dependency, the combination dependency, the definition dependency, and the declaration dependency. Five dependency structures belong to the structure of strong dependency. The structure of strong dependency contains the following structures: the single dependency, the tightly coupled, the closed-loop dependency, and the open-loop dependency.

After preprocessing the file dependency graph, we cluster code elements based on the distance between code elements. The distance is calculated based on the dependence intensity and the similarity of the directory.

Dependency Intensity(DI) represents the degree of the closeness between two code elements, and it depends on the dependency type and the number of dependencies. The DI between the code element x and the code element y is calculated as the following formula.

$$DI_{ab} = \frac{\sum_{i=1}^{N} \alpha_i Num(i, A, B)}{ln(LOC_A)} \tag{1}$$

where N represents the number of dependency types between a and b, i represents the i_{th} dependency type, $Num(i, a, b)$ represents the number of i dependency between a and b, α_i represents the weight of the dependency type.

Directory similarity(DirSim) describes the degree of the similarity between directories. If two code elements are in the same directory, the directory similarity is 1. The directory similarity between the code element a and the code element b is calculated as the following formula.

$$DirSim(a, b) = \frac{|Dir(a)| \cap |Dir(b)|}{|Dir(a)| \cup |Dir(b)|} \tag{2}$$

where $DirSim(a, b)$ is directory similarity between the code element a and the code element b, the $Dir(a)$ represents the directory path of a, $|Dir(a)|$ represents the directory depth of a.

Element distance(ET) is used to describe the distance between two code elements. We consider the distance between code elements based on the directory similarity and the dependency strength. The ET between the code element a and the code element b is calculated as the following formula.

$$ET(a, b) = DirSim(a, b) * DI_{ab} \tag{3}$$

where $ET(A, B)$ is the code element A and the code element B, $DirSim(A, B)$ represents the directory similarity between A and B, and DI_{AB} represents the dependence intensity between A and B.

The clustering efficiency is low when it deals with large amounts of code elements. Therefore, ARO first performs the K-center clustering algorithm to reduce the number of code elements. The K-center clustering algorithm contains four steps. Firstly, the code elements are sorted according to the sum of fanin and fanout of them. Secondly, the top K code elements are the clustered centers. Thirdly, the nearest code elements of them are clustered into centers. Fourthly, if the number of code lines is less than the threshold, ARO performs the first three steps iteratively. The purpose of performing the K-center algorithm contains the following aspects. Firstly, the number of code elements is reduced. Secondly, code elements are clustered around the K centers, and the K centers are the core code, that is, code elements are clustered around the core code.

After performing the K-center algorithm, the K clusters are used to construct the $k * k$ structure, and $L(K)$ denotes the level of the k_{th} cluster, and the distance between the cluster (r) and the cluster (s) is represented as $d[(r),(s)]$. The details of the clustering process are as follows.

1) There are k clusters in the structure D, and each cluster is composed by a code entity. Let the number m is 0, and the $L(m)$ is 0.

2) The minimum of the distance in the D is represented as min, and the cluster is (r) and (s).

3) (r) and (s) are clustered together into a new code entity (r,s). Let the number m is $m+1$, and the $L(m)$ is $d[(r),(s)]$.

4) The D is updated, and the columns and rows of (r) and (s) are deleted. The new cluster (r,s) is added into D. Let the distance between cluster $k-1$ and (r,s) is $mind[(k),(r)],d[(k),(s)]$.

5) Step 2 to step 4 are performed iteratively until the number of code lines of the code elements is greater than the threshold.

The K-center-hierarchy clustering algorithm is shown in Algorithm 1.

Algorithm 1 The K-center-hierarchy clustering algorithm

```
Input:
    Preprocessed file dependency graph $PFDG$
Output:
    Component dependency graph $CDG$
 1: Procedure Cluster(PFDG graph)
 2: for each $node \in nodesets$ do
 3:     if node is a central node then
 4:         process the next node
 5:     end if
 6:     if $node.scale > Value$ then
 7:         process the next node
 8:     end if
 9:     for j=i+1;j<nodesets.num;j++ do
10:         if $min > s(i,j)$ then
11:             min=s(i,j)
12:         end if
13:         select the min distance s(i,r)=min
14:         combine (i)node and (r)node into one cluster
15:         Update graph by updating rules(including delete r node and update
            nodesets)
16:         if node is the latest node then
17:             break
18:         else
19:             i=i-1
20:             Until all the clusters beyond the cluster size
21:         end if
22:     end for
23: end for
```

C. Optimizing architecture

The first step of optimizing architecture is clustering code elements based on the structural similarity. The structural similarity between the code elements is equal to the average of the fanin and the fanout. $RelativeSim(a,b)$ denotes the structural similarity between the code element a and the code element b. The formula is as follows:

$$RelSim(a,b) = \begin{cases} \frac{C\sum_{i=1}^{|O(a)|}\sum_{j=1}^{|O(b)|} ReleSim(O_i(a),O_j(b))}{|O(a)||O(b)|} & a \neq b \\ 1 & a = b \end{cases} \tag{4}$$

where $O(a)$ denotes the fanout of a, the parameter C is a damping factor. If $O(a) = O(b) = A$, then $RelSim(a,b) = C * RelSim(A,A) = C$, so $C \in (0,1)$. If two the structural similarity between two nodes is higher than the threshold, the two nodes are clustered into a new node.

The second step is renaming components. The file name and the directory name are set by developers based on the function of the source code, so we rename components based on their corresponding directory names. The renaming process contains the following steps: a) Identify the corresponding component of each directory. b) All files are traversed to find out whether there is a specific directory contains most files of the component, if there is, then the component is named the directory name. c) All code elements are traversed to find out whether there are three generations of relatives that contain most of the files in the component, if there is, then the component is named as their common grandparent directory.

IV. Experiments and Evaluation

In this section, we evaluate ARO by comparing it with some related methods to answer the following research questions:

RQ1: Does our method improve the accuracy of the recovered architecture?

RQ2: Does our method improve the efficiency of the recovered architecture?

A. Accuracy evaluation

MoJoSim is an indicator of evaluating the similarity between the recovered architecture and the ground-truth architecture, and it is used in much related work [11], [12]. In this paper, we use it to evaluate the accuracy of ARO. MoJoSim is calculated as follows.

$$MoJoSim(RA,GA) = (1 - \frac{mno(RA,GA)}{n}) * 100\% \tag{5}$$

where RA is the recovered architecture, GA is the ground-truth architecture, $mno(A,B)$ is the minimum number of *Move* and *Join* operations of converting A to B, n is the number of clustered code elements. If MoJoSim is 100%, it indicates that the two architectures are the same, and 0% indicates that the two architectures are completely inconsistent.

Table I shows the MoJoSim range of the nine methods and our method.

TABLE I
THE MOJOSIM RANGE OF EACH METHOD

Data source	Method	MoJoSim
Our paper	Our method	0.55-0.75
Bittencourt and Guerrero et al. [13]	EQ	0.20-0.60
	KM	0.40-0.80
	MQ	0.30-0.70
	DSM	0.35-0.75
Wu et al. [17]	CL90	0.40-0.48
	CL75	0.45-0.52
	ACDC	0.28-0.35
	SL75	0.10-0.15
	SL90	0.07-0.10

As shown in Table I, our method has a higher MoJoSim value than other methods. DSM, KM, and our method cluster elements based on the distance, but only KM and our method cluster elements around the core code. In a word, the architecture recovered by our method is closer to the ground-truth architecture, so the accuracy of our method is higher than the above methods.

B. Efficiency evaluation

Bittencourt et al. [12] proposed a cluster-based architecture recovery method, then Michele et al. [13] introduced fold-in and fold-out based on Bittencourt's method. Both of the above methods have been widely recognized, so we compare our method with the above two methods to evaluate efficiency of ARO.

`PostGreSQL` is an open source project, and it is often used as an experimental case for architecture recovery. We choose 30 versions as the experimental cases. The time consumed of the above three methods is shown in Figure 1.

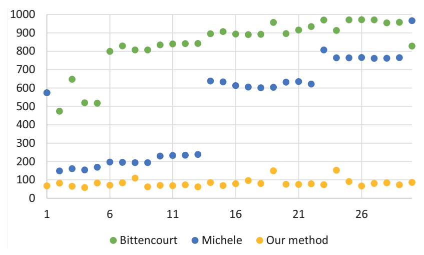

Fig. 1. The time consumed of the three methods

As shown in Figure 1, the average time consumed of the above three methods is 835.288 seconds, 495.568 seconds and 81.814 seconds, respectively. Bittencourt's method takes the longest time, and our method takes the shortest time. In a word, compared to the two methods, our method has higher efficiency.

V. CONCLUSION AND FUTURE WORK

In this paper, we propose a method for recovering and optimizing software architecture based on source code and directory hierarchies to improve accuracy and efficiency. ARO extracts specific information about the implementation of architecture from source code and directory hierarchies to improve accuracy. Then ARO improves the efficiency by preprocessing the file dependency graph and the K-center-hierarchy algorithm. Finally, ARO optimizes architecture. Experimental results indicate that compared with the representative architecture recovery methods listed in this paper, the architectures recovered by ARO has higher accuracy with higher efficiency.

In our future work, we will extract more information about architecture to improve accuracy, such as the compiled build files, architecture documents, and other files.

ACKNOWLEDGEMENTS

This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1003902, in part by the Cooperation Project with Huawei Technologies Co., Ltd., under Grant YBN2016020009, and in part by National Natural Science Foundation of China under Grant 61872078, Grant 61572126, and Grant 61402103.

REFERENCES

[1] Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, and Samuel Kounev. Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. *IEEE Transactions on Software Engineering*, 41(2):157–175, 2015.
[2] Ricardo Britto, Daria Smite, and Lars Ola Damm. Software architects in large-scale distributed projects: An ericsson case. *IEEE Software*, 33(6):48–55, 2016.
[3] Koziolek, Heiko, Schlich, Bastian, Becker, Steffen, Hauck, and Michael. Performance and reliability prediction for evolving service-oriented;software systems. *Empirical Software Engineering*, 18(4):746–790, 2013.
[4] Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. *Using fold-in and fold-out in the architecture recovery of software systems*, volume 24. 2012.
[5] Xianglong Kong, Bixin Li, Lulu Wang, and Wensheng Wu. Directory-based dependency processing for software architecture recovery. *IEEE Access*, 6:52321–52335.
[6] Liu Jing, Zhiming Lui, Xiaoshan Li, Jifend He, and Yifeng Chen. Towards the integration of a formal object-oriented method and relational unified process. *Software Evolution with Uml & Xml*, pages 101–133, 2005.
[7] Fritz Solms. Experiences with using the systematic method for architecture recovery (symar). In *South African Institute for Computer Scientists and Information Technologists Conference*, pages 170–178, 2013.
[8] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.
[9] Kamran Sartipi. Software architecture recovery based on pattern matching. In *International Conference on Software Maintenance*, pages 293–296, 2003.
[10] H Sajnani. Automatic software architecture recovery: A machine learning approach. In *IEEE International Conference on Program Comprehension*, pages 265–268, 2012.
[11] Thibaud Lutellier, Devin Chollack, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *IEEE/ACM IEEE International Conference on Software Engineering*, pages 67–69, 2015.
[12] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *European Conference on Software Maintenance & Reengineering*, pages 251–254, 2009.
[13] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. A probabilistic based approach towards software system clustering. In *European Conference on Software Maintenance and Reengineering*, pages 88–96, 2011.