An approach for Software Architecture Refactoring Based on Clustering of Extended Component Dependency Graph

Zhong Lin_Hui, Xu Le, Ye Mao_sheng, Zheng Yi, XieBing

shiningto@yahoo.com.cn

(① School of Computer Information and Engineering, JiangXi Normal University, Jiangxi 330022, China)
(② Nanchang Hangkong University, NanChang ,Jiang Xi , 330063 )
(③ School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract: For improving the evolvability of software architecture, the paper proposes a software architecture refactoring strategy based on extended clustering of component dependency relation, which consists of logical relation and evolution relation among components. By using the graph clustering algorithm, the software architecture can be restructured according to the software quality of “high cohesion and low coupling” under the control of our refactoring algorithm. Moreover, an example is shown for explaining its usability.

Keywords: software evolution, evolution information, software architecture, software refactoring

I. INTRODUCTION

Refactoring is “the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [1]”. Now software refactoring can be referenced to any software techniques that reduce software complexity by incrementally improving the internal software quality [2]. Refactoring can be done at any level of software such as software architecture, which called software architecture refactoring[2]. Software architecture refactoring can improve software quality such as evolvability by modifying the relation between component and component itself. In [8], Philpps proposed a approach for software architecture based on graph by introducing graph rewrite rules, which can hold the behavior between components. Also in software product line, refactoring can support the software family evolve by evolving the variant components into common components [9].

Based on our previous research on component model supporting evolving[3][4], in this paper we take software architecture as the topology of components in complex component, and propose a strategy for software architecture refactoring using static relation and evolving relation among components by clustering algorithm. So we firstly propose the concept of extended component dependency relation in Section 2, then introduce the process of software architecture refactoring and its key algorithms using the architecture of Coomplier as example in Section 3, finally the conclusion is given in Section 4.

II. EXTENDED COMPONENT DEPENDENCY RELATION GRAPH

Component relation can be described by the Component Dependency relation graph, in which the node refers to the software component and the edge refers to the dependency between the components, the weight of the edge can be used to denote the degree of the dependency. Often the relation is static such as logic relation, but in the paper, we also consider the evolution dependency relation and unify them into a graph called extended component dependency relation graph (ECDRG). Some definitions is given below.

Definition 1: ECDRG=<C, E>. The C is the collection of components in the software system; E⊂C×C, E is the collection of the relation between components, which consists of logic dependency relation(such as composite relation and delegate relation between interface[5][6].) and evolving dependency relation(the relation among the change history, discussed in Definition 2). For each edge such as e, e∈E, W(e)denotes the weight of the e, W:C×C->N+.

Definition 2: Component Evolving Dependency (CED). Let C is the collection of all the components in software architecture. If the component c1, c2…ck changed together in the same versions (denote as V) among the history of the software architecture, then we call the changed components have Component Evolving Dependency (CED) and the weight of the CED is |V|(the number of the V).

Definition 3: Component Evolving Dependency Graph (CEG), is a directive graph with weight. CEDG=<C, E>, C= CED.ComSet, satisfies

∀(c1,c2 : c1∈C, c2∈C : (c1,c2)∈ E ∧ (c2,c1)∈ E)

The weight is W, W: V×V->CED.Value. That is, every two component c1 and c2, there are two edges (<c1,c2> and <c2,c1>)between them.

The evolving dependency relation can be classified into single changed evolving dependency relation and multiple changed evolving dependency relation. In the follow section, we will discuss how to get the multiple changed evolving
dependency using data mining technology. Firstly, we discuss the technology of getting single changed evolving dependency relation, which is the basic of the multiple changed evolving dependency relation.

1. Single changed evolving dependency relation

\[ R_i = \langle \{ C_1, C_2, \ldots, C_k \}, \{ V_i \} \rangle \]

denotes the component \( C_1, C_2, \ldots, C_k \) changed together from Version i-1 to Version i of the software architecture.

Single changed evolving dependency relation can be got by calculating the version number of component in the extended software component language-xJBCDL[3]. If the components changed in the \( i^{th} \) version software architecture, then the changed components belong to the Single changed evolving dependency relation. The Algorithm is shown in Algorithm 1.

**Algorithm 1:**

Suppose the ith Single changed evolving dependency relation is stored in Collection \( R_i \), \( R_i = \emptyset \); The ith version xJBCDL of the software architecture is marked as xJBCDLi.

```plaintext
FOR every Component C in xJBCDLi DO
    Get the version number of the C into variable t1; Search the version number of the C in xJBCDLi, and store it into variable t2;
    IF \( t1 = t2 \) THEN
        \( R_i = R_i + C; /* add the Component C into the Single changed evolving dependency relation */
    ENDIF
END FOR
```

2. Multiple changed evolving dependency relation.

Given an change sequence \( R_i, \ldots, R_j \), the multiple changed evolving dependency relation can be defined as \( R_{i,j} = \langle \{ C_i, C_j, \ldots, C_k \}, \{ V_i, \ldots, V_j \} \rangle \), which means the software architecture version \( V_j, V_i \in \{ V_i, \ldots, V_j \} \), if software architecture changed from \( V_k \) to \( V_l \), then Component \( C_i, C_j, \ldots, C_k \) changed together.

In this paper, we use the concept lattice [13] to get the multiple changed evolving dependency relation. Concept lattice is a formal approach to cluster the object using the user-defined attributes. So we take the component of the software architecture as Object, and the version number as the attributes. Also if the component belongs to the Single changed evolving dependency relation of the corresponding software architecture version, and then there is a value in the context of the concept lattice. An example is shown in figure 1, in which the white square denotes a version of the component, if the component changed, then marks it as black square.

![Figure 1. The change history of software architecture](image)

The corresponding concept lattice is:

```
\[ \langle \{ C_1, C_2, C_3, C_4, C_5, C_6 \}, \{ V_i \} \rangle \]
```

The concept in Concept lattice denotes a component evolving dependency relation. We take those concepts as the candidate multiple changed evolving dependency relation, whose number of component collection or version collection is bigger than one. Furthermore we use the heuristic information to help chose the most intensity evolves dependency relation based on measurement. Once the changed evolving dependency relation is selected, we can make the changed evolving dependency relation into the changed evolving dependency relation graph. The heuristic information is measured as followed:

Supposed component evolving dependency relation is denoted as \( \langle \{ C_1, C_2, \ldots, C_m \}, \{ V_1, V_2, \ldots, V_n \} \rangle \), and the number of all components is \( M \), the number of the versions is \( N \), then the heuristic information is measured as \( \sqrt{(m/M)^2 + (n/N)^2} \). Using the formula, the bigger of the value, the
more intensity of the component evolving dependency relation, in which the components have the higher probability to become a compose component.

III. THE PROCESS OF SOFTWARE ARCHITECTURE REFACTORIZING BASED ON ECDRG

The “high cohesion and low coupling” is a good indicator for software quality such as evolvability. If the component itself has high cohesion and low coupling between components, then the change effect can be controlled within a limited range and the designer can know what the change has effect on. So in the follow, a refactoring approach is given by using the clustering approach.

A. criterion of clustering

Software architecture refactoring is a process of restructuring software architecture configuration for improving the quality. In this paper, software architecture is clustered based on extended component dependency graph, and lots of sub-graphs are created, which have “high cohesion and low coupling” character. Finally, software architecture refactoring is proceeding based on those sub-graphs.

Lots of graph clustering algorithms can be used for portioning software such as pattern-driven clustering algorithm ACDC [10], hill climbing clustering algorithm NAHC and SAHC[11], and genetic clustering algorithm[12]. In this paper, we use the Bunch graph clustering algorithm[11] and its tools as our research basis. In his research, MQ is used to represent the quality of the graph, which value is between -1 and 1. The bigger the MQ is, the higher the quality of the graph fits the “high cohesion and low coupling”. The formula of MQ is

\[ MQ = \begin{cases} 
\frac{1}{k} \sum_{i=1}^{k} A_i - \frac{1}{k(k-1)} \sum_{i,j=1}^{k} E_{i,j} & \text{if } k > 1 \\
A_1 & \text{if } k = 1
\end{cases} \]

In which, the \( A_i \) represents the inner connectivity of the ith sub-graph, which is affected by Ni and \( \mu_i \). Ni is the number of the nodes of the ith sub-graph, and \( \mu_i \) is the edge connectivity between the nodes. \( E_{i,j} \) represents the connectivity between the ith sub-graph and the jth sub-graph, \( E_{i,j} \) is affected by the number of values \( N_i, N_j \), and the edge connectivity between the ith sub-graph and jth sub-graph. The formula of \( A_i \) and \( E_{i,j} \) is follow:

\[ A_i = \mu_i / N_i^2 \]

\[ E_{i,j} = \begin{cases} 
0 & \text{if } i = j \\
\frac{\epsilon_{i,j}}{2N_i/N_j} & \text{if } i \neq j
\end{cases} \]

Using the Bunch tools, we can input a set of triple relation \(<v_1, v_2, \text{weight}>\), which means there is a dependency between Node \( v_1 \) and Node \( v_2 \), and weight denotes the degree of the dependency. After processing, Bunch tools can output a set of sub-graphs which is composed of nodes.

B. The algorithm of software refactoring based on Clustering

After portioning it with Bunch graph clustering algorithm on the extended component dependency relation, then software architecture refactoring can be done using the following algorithm [7].

Algorithm 2:

Input: given the software architecture of a software system described by xJBCDL, and the extended component dependency relation graph (ECDG).

Output: the xJBCDL of the software system after refactoring

Begin

The ECDG is portioned into the collection of sub-graphs is collection of Gset.

① Firstly consider all of the sub-graph \( G \) in Gset, for every edge in G, \( e=(v_1, v_2) \), if e is the kind of “evolving dependency relation”, then

If \( v_1 \) and \( v_2 \) belongs to the same component and have the same level, then calls “same level component consolidation operator”;

If \( v_1 \) and \( v_2 \) respectively belongs to the two same level composite component then calls “same level component consolidation operator”

If \( v_1 \) and \( v_2 \) is at the different level, then calls “different level component consolidation operator”;

② For every two sub-graphs \( G1,G2, G1 \in \text{Gset} \), \( G2 \in \text{Gset} \):

Begin

For every edge between \( G1 \)and \( G2 \), \( e=<v_1, v_2> \), \( v_1 \in G1 \), \( v_2 \in G2 \)

Begin

If the e is the kind of “compose relation”, then elevators the v2 using elevator operator and marks the v2 as “needs consolidation” node;

If the e is the kind of “delegate relation”, then elevators the v1 and v2 using elevator operator;

End

End

Consolidates all of the “needs consolidation” node in G using “same level component consolidation operator”

End

IV. AN EXAMPLE

Supposed a compiler can use different semantic analyzer such as LR, LL to analyze software program, which software
architecture is shown in figure 2. In which, Component LL, LR1 and LR2 can analyze program using their corresponding state machine; Component SemLR1, SemLR2 and SemLL is used to store the semantic translate rules; Component TLRTuning is responsible for configuring and calling state machine such as Component LL and semantic translate rules such as SemLL. Notices the state machine must be used together with corresponding semantic translate rules.

Suppose the software architecture of the complier evolves for several versions. According to the algorithm of computing evolving relation mentioned in Section 2, the Multiple changed evolving dependency relation can be: \(<\{\text{LR1, SemLR1}\},\{v_1, v_2, v_3\}>,\,<\{\text{LR2, SemLR2}\},\{v_1, v_2, v_3, v_4\}>,\,<\{\text{LL, SemLL}\},\{v_2, v_3, v_5\}>>;

Figure 2. the software architecture of the Compiler supporting multi-Grammar

After adding the evolving dependency relation and clustering it using Bunch Graph clustering algorithm, the software architecture of complier is shown in figure 3, in which the dotted line represents the evolving dependency relation and the components surrounded by the dashed line is the candidate for refactoring. Using the domain knowledge Component LR1 and SemLR1 are used together for proceeding the LR1 grammar, so we can refactor them as a new compose component named TLR1Manager. Also Component LR2and Component SemLR2 can be refactored as TLR2Manager; Component LLand Component SemLL can be refactored as TLLManager.

Figure 3. after refactoring based on clustering extend component dependency relation graph

V. CONCLUSION

In component based software development, the quality of software architecture is important for the final software. In this paper, we research the effort of the logic dependency relation and the evolving dependency relation on the evolvability of the software architecture, and propose an approach for software architecture refactoring based on the clustering of the extended component dependency relation graph. Final, we take the complier as an example to verify our approach effective.

Using above algorithm for software architecture cannot always guarantee the refactored software architecture having the same semantic as before, the architecture designer needs to reason its rationality using domain knowledge. But using our approach of adding evolving dependency into component dependency graph, it can enrich the connectivity between components and increase the accuracy of the clustering.

Acknowledgment

Thanks Professor Sao Wei_Zhong, Professor Zhang Lu of the SEI in Peking University - and is supported by science technology project of Jiangxi education government (No. GJJ09140), National Natural Science Foundation of China (No.60403015) and Natural Sciences Foundation Projects of Jiangxi

CONFERENCES