Towards the Software Evolution Recovery at the Level of Software Architecture

YanXia Zhu¹, LinHui Zhong¹*, LiJuan Fu¹*, ShuHe Ruan¹, Jing Xu², Haitao Ye³

¹College of Computer and Information Engineering, Jiangxi Normal University, Nanchang, Jiangxi, 330022, China
²School of Chinese Language and Literature, Jiangxi Normal University, Nanchang, Jiangxi, 330022, China
³The First Affiliated Hospital of Nanchang University, Nanchang 330006, China

*Corresponding author’s e-mail: shiningto@jxnu.edu.cn; 201940100646@jxnu.edu.cn

Abstract—In the process of software evolution, a lot of evolution history information will be stored in software repository. Because some open source repositories (such as GitHub) only record the version of every source file, it is difficult for people to comprehend and manage the evolution of software component and software architecture. In the paper, we propose the concept of software evolutionary binary tree to present the evolution history of software architecture and its components, propose a method to recover them by software architecture recovery technology as well. The experimental results of four open source software (Cassandra, Hbase, Hive and Openjpa) demonstrate that the method is effective.

1. INTRODUCTION

In order to effectively support the software architecture-centered and component-based software development methods, many researchers advocate the combination of software architecture and software configuration management [1] and propose the idea of software evolution management model to version the software architecture and component, etc. [2]. However, those models still face many software evolution problems. It is difficult to track and comprehend the history of the software components and software architecture because the software system stored in GitHub only records the version of every source file.

The management of evolution process usually tracks and manages the change of software elements and the change among elements through application software configuration management model and other corresponding tools. For example, an evolution management model which can support multi-layer software architecture and track the changes of software in specification, implementation and deployment was designed [3]. Zhong combined the definition of version information with component model and software configuration management model, and proposed a component model which can support component-based software evolution to realize the evolution management of component-based software [4].

For those software systems which do not employ software configuration management systems for storage, many researchers proposed to track the relationship between software elements by using some techniques like static analysis, text analysis and visualization analysis. For example, in order to find trajectory patterns of code changes, Jiang grouped and clustered the changes [5]. Servant proposed two central themes [6,7].
Different from such methods mentioned above, this paper puts forward a method which could analyze the evolution history of software systems by constructing evolutionary binary trees. Firstly, the background of this paper and the related works about software evolution are introduced. Then, the process of software architecture recovery is described in section 2, the concept and construction rules of software evolutionary binary trees are involved in section 3, and experiments are showed in section 4.

2. THE PROCESS OF SOFTWARE ARCHITECTURE RECOVERY
In component-based software development, a large software system is composed of software components. We can regard the components inside the software system as atomic components and take the system as a composite component which consists of atomic components and the dependencies among them.

There are a plenty of methods can reverse the software architecture. For example, Kouroshfar proposed a method to reverse the system architecture into intermediate results as packages, clusters, modules, etc.

2.1 Atomic Components recovery
Every system version can generate multiple clusters and dependencies among classes in the cluster through inversion. When generate all atomic components by applying the class dependency of a system version and the source codes of it, firstly it’s proper to reverse each cluster into an atomic component, rename the component name as its corresponding cluster name, and add all the classes in the cluster into the class set of the atomic components. Next, we should construct functional relationships of Provides and Requires between atomic components according to the dependency relationships among classes in the cluster, and respectively store them in Provides and Requires. Finally, we can obtain the multi-dimensional attribute information of atomic components according to system’s source code and store them in corresponding atomic components, so that the evolutionary trees of the atomic components can be constructed later in the light of the multi-dimensional attribute information.

2.2 Composite Component recovery
After all atomic components in a system version being reversely generated, we should use them to regenerate a composite component which can represent the architecture of the system. Firstly, a system could generate a composite component, so the composite component name should be renamed to its corresponding system name, Next, an instance from each atomic component should be generated and added into the sets Instance of its corresponding composite components. Then, the functional dependency relationships among atomic component instances of the composite components should be constructed according to the functional relationships of Requires in atomic components, and added into Relation in composite components. Finally, the multi-dimensional attribute information of these composite components should be calculated according to the multi-dimensional attribute information of all atomic components and stored in their corresponding composite components, so that the evolutionary trees of the composite components can be constructed later.

3. THE CONSTRUCTION METHOD OF EVOLUTIONARY BINARY TREE
The evolution relationships between software system versions can be represented by evolutionary trees. Data structure tree and binary tree can be converted to each other equivalently [8]. Hence, this paper depicts the relationships by evolutionary binary trees for convenience.

3.1 Evolutionary Trees and Evolutionary Binary Trees
Definition 1. Evolutionary tree is a tree T= (Root, F). Root is a root node, F is the forest composed of m (m ≥ 0 ) subtrees, i.e. F= (T₁, T₂, ..., Tᵢ, ..., Tₘ), and Ti is the i-th subtree of T.

In the context of software evolution, if there was a path from the non-leaf node Nₘ of the evolution binary tree to a leaf node Nₙ, then there is an evolutionary branch from Nₘ to Nₙ or several branches from Nₘ to other leaf nodes.
Definition 2. The evolutionary binary tree is a kind of evolutionary tree with special meaning, i.e. $T=(\text{root}, L, R)$. $L$ and $R$ respectively mean the left and right child nodes of the root node. As stipulated in this paper:

1) The leading branch is the evolutionary branch which starts from any non-leaf node $N_m$ along with the left child node to a leaf node $N_n$, and the side branch is the evolutionary branch which starts from an inner node $N_m$ to any leaf node $N_k$ of a sub-binary tree with the root the right node of $N_m$. As shown in Figure 1, Leading Branch is denoted with the red dotted box, and the blue dotted box indicates Side Branch.

2) In software evolution, a new branch often starts from one version $V$ to another. In order to clearly distinguish different evolutionary branches, we define the copy node of $V$ as a virtual node and take it as the first version of the new branch. Any node is taken as $V_{1.0}$, any right child node $V_{1.0.0}$ existed is a copy of $V_{1.0}$, i.e. $V_{1.0.0}=\text{Clone}(V_{1.0})$. The cloned node is called a virtual node, and the no-cloned ones are real nodes.

![Figure 1. Evolutionary Binary Tree](image)

3.2 Generative Rule of Evolutionary Binary Tree

The rule of constructing an evolutionary binary tree is to generate real nodes in the generation order of component versions, and then inserting them into an evolutionary binary tree. Here are the main steps to insert the real node $Q$ generated by a component version into the evolutionary binary tree: we must calculate the similarities between all the real nodes in the evolutionary binary tree and $Q$ at first, and select the real nodes of the tree according to the level of similarity from high to low as the node $P$ to be inserted, judge by the similarity degree between $Q$ and $P$, and manage to insert $Q$ into an evolutionary binary tree according to the judgment results in the way of inserting $Q$ into the left child of $P$ while regarding it as a major evolutionary branch or inserting it into the right child of $P$ while regarding it as a side branch. Here are five cases occurred during the inserting process:

1) The evolutionary binary tree is empty, $Q$ is taken as the root node of the tree and its version number was taken as the tree’s initial version number;

2) The tree is not empty:

1) If $Q$ and $P$ were the same component version, then it’s proper to merge $Q$ and $P$, add $Q$’s baseline into $P$’s baseline sequence, and rename $Q$’s version number as same as $P$’s;

2) If $Q$ and $P$ were different component versions with high similarity, and $P$ doesn’t have a left child node, then $Q$ will be taken as the left child node of $P$, and $P$’s version number plus 1 will be taken as $Q$’s;

3) If $Q$ and $P$ were different component versions with high similarity but $P$ has left child node, a next similar real node will be re-selected as the new node $P$ to be inserted, then the results will be re-judged;

4) If $Q$ and $P$ were different component versions with low similarity and $P$ does not have a right child node, a virtual node $\text{Clone}(P)$ will be constructed then taken as the right child node of $P$ while $Q$ will be taken as the left child node of $P$, $P$’s version number with an added “.0” will be taken as $\text{Clone}(P)$’s, then $\text{Clone}(P)$’s version number plus 1 will be taken as $Q$’s;
5) If Q and P were different component versions with low similarity but P has a right child node, a virtual node clone(P) will be constructed after finding the root node R of the rightmost side branch of P, and clone(P) will be taken as R’s right child node while Q will be taken as clone(P)’s left child node while R’s version number with an added “.0” will be taken as clone(P)’s, then clone(P)’s version number plus 1 will be taken as Q’s.

4. EXPERIMENTS
The source codes of the four open source software systems and the experimental data provided by third parties [9] were used as our experimental basis. The specific version information of the four systems and the number of atomic components were reversely generated before shown in Figure 2. The system version number was setted in GitHub already, and version baselines were rearranged according to the open source system versions (starting at 1); In the case that we only have the source codes of multiple system versions of the four open source systems above and the corresponding class dependency graphs, the evolutionary binary trees of an open source system’s real system versions can be artificially constructed, but the evolutionary binary trees of an atomic component’s real component versions can’t be constructed. Hence, this paper only shows the tree edit distance between the evolutionary binary trees of composite components and that of real system versions.

Similarity threshold should be set when atomic and composite components were reversely generated through the rules of generating evolutionary binary trees. Different thresholds may lead to discrepancies in tree structure. In order to compare the differences between the reverse-generated trees and the real trees of system versions, the tree edit distance algorithm [10] was utilized to calculate the minimum cost of the converting process of the two kinds of trees. In the following two experiments, the changes of the minimum costs under the different values of similarity thresholds were demonstrated.

The changes of the minimum costs under the different values of similarity thresholds are presented. We depicted the variation of the edit distances between the evolutionary binary trees of composite components generated from the four open source systems and the real trees of system versions with the change of similarity thresholds as shown in Figure 2.

![Figure 2. Corresponding changes of tree edit distance in different similarity thresholds](image)

It can be seen from Figure 2 that when similarity thresholds were set within certain ranges, the edit distances between the generated evolutionary binary trees of composite components and the real ones of system versions were the same, in other words, the generated evolutionary binary trees of composite components were the same when the similarity thresholds within the ranges. In addition, with the increasing of similarity threshold value, the edit distances between the generated evolutionary binary trees of composite components and the real ones of system versions were in a step-style growth trend. And when the similarity threshold was set to 1.0, the edit distance reaches its maximum value.
Experiments show that when the similarity thresholds are smaller, the edit distances between the evolutionary binary trees of composite components and that of real system versions are shorter, that is to say, the two kind of trees are becoming more similar. The reasons may be that: (1) in our experiments, the amount of system versions are insufficient; (2) in reality, the evolutionary binary tree of a real system version looks similar as a tree with a single branch in structure because software system is generally updated on the basis of the previous system version; (3) the multi-dimensional attributes selected as measurement indicators are not representative when measure the similarity of component versions.

5. CONCLUSION
The combination of software architecture recovery and software evolution analysis can effectively support the development and management of component-based software. On the basis of the proposed concept and generation rules of software evolutionary binary tree, this paper obtains the evolutionary relationships among software system versions by employing software architecture recovery technology, and achieves the purpose of analyzing software evolutionary history by clarifying these evolutionary relationships in order to make up for the deficiencies of current software evolutionary management models. What’s more, the evolutionary histories of those four open source software systems and that of components were acquired through experiments by the method proposed. These efforts are favorable for the auxiliary management of component software’s development.

ACKNOWLEDGEMENTS
This research was supported by the National Natural Science Foundation of China (61462040) and the Scientific Research Project of Jiangxi Normal University (JXSDJG2044).

REFERENCES
[1] Westfechtel B, Conradi R. Software architecture and software configuration management[M]. Software Configuration Management. Springer, Berlin, Heidelberg, 2003: 24-39.
[2] Gergic J. Towards a versioning model for component-based software assembly[C]//Software Maintenance. ICSM. Proceedings. International Conference on. IEEE, 2003: 138-147.
[3] Mokni A, Urtado C, Vauttier S, et al. A formal approach for managing component-based architecture evolution[J]. Science of Computer Programming, 2016, 127: 24-49.
[4] ZHONG Lin-Hui, XIE Bing, SHAO Wei-Zhong. Supporting Component-based Software Development by Extending the CDL with Software Configuration Information [J]. Journal of Computer Research and Development, 2002, 39(10): 1361-1365.
[5] Jiang Q, Peng X, Wang H, et al. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes[C]. Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, 2015: 361-370.
[6] Servant F, Jones J A. Fuzzy fine-grained code-history analysis[C]. Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017: 746-757.
[7] Servant F, Jones J A. History slicing: assisting code-evolution tasks[C]//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012: 43.
[8] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. Data Structures and Algorithms in Java[M]. USA: John Wiley & Sons, 2014:308-350.
[9] Kouroshfar E, Mirakhorli M, Bagheri H, et al. A Study on the Role of Software Architecture in the Evolution and Quality of Software[C]. Proceedings of the 12th Working Conference on Mining Software Repositories. IEEE Press, 2015: 246-257.
[10] Zhang K, Shasha D. Simple fast algorithms for the editing distance between trees and related problems[J]. SIAM journal on computing, 1989, 18(6): 1245-1262.