2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Extensions of Component Rank Model by Taking into Account for Clone Relations

Reishi Yokomori*, Norihiro Yoshida†, Masami Noro* and Katsuro Inoue‡
*Department of Software Engineering, Nanzan University, Nagoya, Aichi, 466–8673, Japan
Email: yokomori@se.nanzan-u.ac.jp, yoshie@nanzan-u.ac.jp
†Graduate School of Information Science, Nagoya University, Nagoya, Aichi, 464–8601, Japan
Email: yoshida@ertl.jp
‡Graduate School of Information Science and Technology, Osaka University, Suita, Osaka, 565–0871, Japan
Email: inoue@ist.osaka-u.ac.jp

***Abstract*—The size of software increases in recent years, and the number of classes and relationships between classes are also increasingly complicated. We have a large interest about the way to classify a great deal of components effectively. Our research group proposed a component rank model in the past, and the model calculates an evaluation value of each component by using component graph that represents use-relations between components. In this paper, we suggest a method to reflect code clone's relation on the component rank model. In the extended model, code clone's relations are reflected on the component graph by merging components that have similar code fragments. When we focus simply on how the evaluation value of each component has changed, component whose evaluation value falls would be mainly a component used by both of the merged components. In the experiment, we applied our method to several open source projects, and confirmed that the change of evaluation value of each component was within the scope of the assumption. And we also searched case examples, and confirmed how much assumed cases existed in the actual software, and a percentage of the reduction of the evaluation value was useful for detecting components related with code clone. Through these experiments, we confirmed that our method is effective to detect components that existing similar code fragments are using.**

I. Introduction

The size of many software increases continuously because of accommodating a lot of additional features. Therefore, the number of classes that compose such software becomes increasing, and relationships between such classes also become increasingly complicated. When implementing new features, developers add new code fragments to the software. Sometimes these code fragments are very similar to the existing code fragments. This is because, implementing a similar feature also requires producing codes that have similar code fragments. Such similar code fragments are called code clone [1], and how to manage the software that contains a lot of code clones becomes a major matter of concern. It is generally desirable to remove or not to create a code clone, however, code clones are sometimes woven with developer's intention. In such case, developers have to share the intension for keeping the maintainability of the software.

Our research group proposed a ranking model for recommending desirable components from a lot of components extracted from a lot of software systems and we call it component rank model [2]. In the model, components are recommended from a viewpoint how many components each component is used by, and a component graph, which represents components and use-relationships between components, is produced, and evaluation value of each component is obtained by repetitive calculation on the component graph.

In this paper, we extend the component rank model by reflecting relationships of code clones between components on the component graph. In our approach, the target component graph consists of one software, and components that have similar code fragments are merged in the graph. And then we re-calculate evaluation value based on the merged graph, and we get two evaluation values of each component, before and after reflection of the code-clone relations. We take particular note of the not merged components, and calculate a degree of the change of the evaluation value of each component. We will discuss how evaluation value of each component may change through the reflection. We consider affected components are used by both of the merged components, and the evaluation values of these components would decrease.

The goal of our research is to propose a system that detects components that are commonly used by similar code fragments. We consider that simply extracting components that are commonly used by similar code fragments would be insufficient bacause we have to consider use relations to the components from other ones. In the proposed method, we can consider a degree of the aggregation of use relations in merging, so we consider a percentage of reduction of the evaluation value would be useful for detecting assumed components.

By extracting components that are truely related to the existing code clone, we can offer a mechanism to prevent a spread of the code clone. For example, we consider a situation of adding a new component to the existing software. When the new component uses component whose evaluation value has decreased, the new component may use the components by the same method as components that have similar code fragments. At the time of using such components, the expanded model can support by warning a possibility of producing a code clone. We consider that this usage supports developers from two viewpoints: the viewpoint of preventing making a code clone easily, and the viewpoint which shows existences of

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.30

IEEE computer society

code fragments that are useful as examples. In this way, we can keep away that the existing code clone derives and increases in various forms. Since differentiated code clones make it more difficult to detect code clones, so our method would be useful for improvement in the quality of software.

Based on the approach, we implemented a system that shows the components whose evaluation value have changed after reflection of code clone relations, and we conducted two evaluation experiments. In the first experiment, we applied our method to source codes of open source projects and confirmed that the evaluation value of components used by several components in the merged components would decrease. In the second experiment, we searched case examples, and confirmed how much assumed cases existed in the actual software, and confirmed that a percentage of the reduction of the evaluation value was useful for detecting assumed components.

In Section II, we introduce backgrounds. In Section III, we discuss how evaluation value of each component may change by merging components. In Section IV, we introduce our implementation. In Section V, we introduce conducted two experiments. Finally, in Section IV, we discuss about the results and future directions and introduce related works.

II. BACKGROUNDS

A. Component Graph

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [3]. We model software systems by using a weighted directed graph, called a *Component Graph* [2]. In the component graph (V, E) , a node $v \in V$ represents a software component, and a directed edge $e_{xy} \in E$ from node x to y represents a *use relation* meaning that component x uses component y.

B. Component Rank Model

Based on the component graph, we proposed Component Rank Model [2]. In the model, the component graph represents a Markov chain, and a movement of software developer's focus on the target software is represented by a probabilistic state transition. The weight of each node at steady state is regarded as the evaluation value of correspondent component, and this model calls the order of the components sorted by the evaluation value component rank of the components. The component rank model was proposed as a part of component search engine, and its application result was reasonable such that very general and core classes were ranked high (significant), and specific and independent classes were ranked low.

This model introduces several definitions to calculate evaluation values (weights) for component graph $G = (V, E)$. Each node v in component graph G has a non-negative weight value $w(v)$ where $0 \leq w(v) \leq 1$. The sum of the weights of all nodes in G is 1, and total weights of nodes are kept as 1.

The following is a calculation process;

1) Set initial weights to each node
   Each node has $1/n$ as an initial weight if target system consists of n components.
2) Repeat calculation **2-1** and **2-2** until convergence.
   2-1 Calculate weights of edges from weights of nodes
   For each edge $e_{ij} \in E$ from v_i to v_j, we define a weight $w'(e_{ij})$ of e_{ij} as following;

   $$w'(e_{ij}) = d_{ij} \times w(v_i)$$

   Figure 1 (a) depicts this definition. Here, d_{ij} is called a *distribution ratio* for edge e_{ij}, where $0 \leq d_{ij} \leq 1$, and if there is no edge from v_i to v_j, $d_{ij} = 0$. The distribution ratio d_{ij} is used for determining the forwarding weights of v_i to an adjacent node v_j. For each node which has any outgoing edges, the sum of distribution ratios for outgoing edges is always 1. In the current implementation, if a node a has n outgoing edges, and then distribution ratios of outgoing edges from a are all $1/n$. In the actual calculation, we have to treat the target component graph as a strongly connected graph to guarantee the termination of repeated computation, so we introduce pseudo use relations between all nodes. This is not on topic, so please refer to [2].

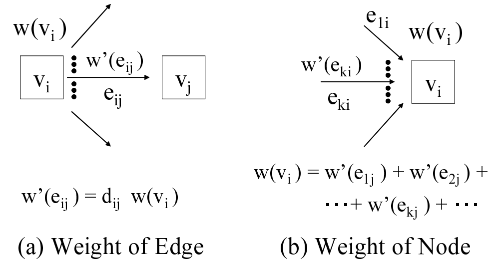

Fig. 1. Definition of weights

   2-2 Re-calculate weights of nodes from weights of edges
   The weight of a node v_i is re-defined as the sum of the weights of all incoming edges e_{ki}, such that

   $$w(v_i) = \sum_{e_{ki} \in \text{IN}(v_i)} w'(e_{ki})$$

   Here, IN(v_i) is the set of the incoming edges of v_i. Figure 1 (b) shows this definition.
3) The weight of each node at steady state is regarded as the evaluation value of correspondent component, and components are sorted by the evaluation value.

Figure 2 shows a component graph with evaluation value at steady state. v_1 has two outgoing edges, and weight 0.4 is evenly divided to two outgoing edges with 0.2 each (i.e., $d_{12} = d_{13}$ = 0.5). v_3 has two incoming edges, each with weight 0.2, so that the weight of v_3 is 0.4.

III. REPRESENTING CODE CLONE RELATIONS ON THE COMPONENT GRAPH

A code clone is a code fragment that has identical or similar code fragments to it in the source code [1]. Code clones appear not only in a single component, but also appear between different components. It is generally desirable not to create a

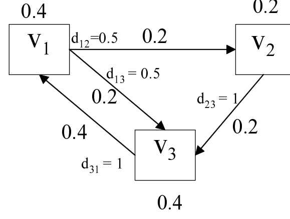

Fig. 2. An example of stable weights assigned to nodes and edges

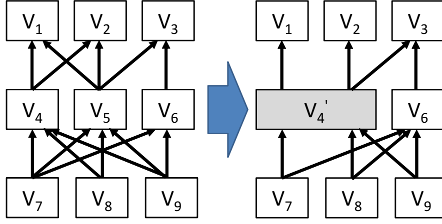

Fig. 3. Extending by merging clone-related components

code clone, however, code clones are sometimes woven with a certain intention. In such case, developers have to share information about the code has similar code fragments, for keeping software maintainability. This is because developers have to review another code fragments to keep its consistency when one of such code fragments should be modified.

In this paper, we extend the component rank model by reflecting relationships of code clones. As a method for reflecting code clone relationships, we merge nodes which have the same or similar code fragments. For examples, we consider a part of component graph in Figure 3. It has 9 nodes (components), and V_4 and V_5 have similar code fragments each other. In this case, we merge V_4 and V_5 into V_4', and draw edges from V_x to V_4' if V_x uses V_4 or V_5, and draw edges from V_4' to V_x if V_4 or V_5 uses V_x, respectively.

In this way, we get two component graphs before and after reflecting code clone relationships, and calculate two component ranks. We focus attention on the components that are not merged, and calculate differences of the evaluation value of each component. In this study, we have an interest about how the evaluation value of each component changes by reflecting relationships of code clones. We examine how evaluation value of each component would change as following:

- Incoming edges to V_4 and V_5 are aggregated to one incoming edge to V_4'. So, the number of outgoing edges from V_7 and V_8 decreases and evaluation value of these edges would increase. So, evaluation value of each incoming edge to V_4' increases.
- Evaluation value of V_4' is a sum of all incoming edges, so evaluation value of V_4' would be larger than the one of V_4 or V_5. However, evaluation value of V_4' doesn't exceed the sum of V_4 and V_5. This is because incoming edges to V_4 and V_5 are aggregated to one incoming edge, and if we consider the situation that x nodes are merged and the number of outgoing edges from a certain node becomes n to $n-x+1$. Evaluation value of each edge become $\frac{n}{n-x+1}$ times, however $\frac{n}{n-x+1}$ is equal or less than x.
- We consider a case of component that is used by both of the merged components. V_1 is used by both V_4 and V_5. Before merging, V_1 gets evaluation value via edges from V_4 and V_5. After merging, outgoing edges from V_4 or V_5 are also aggregated to one outgoing edge, and V_1 gets evaluation value via edges from V_4'. Evaluation value of V_4' doesn't exceed the sum of the one of V_4 and V_5, so the evaluation value of V_1 decreases.
- We consider a case of component that is used by only one of the merged components. V_5 uses V_3 and V_4 doesn't use V_3. Before merging, V_3 gets evaluation value via edge from V_5. After merging, V_3 gets evaluation value via edge from V_4'. The evaluation value of V_4' is larger than the one of V_5, so the evaluation value of V_3 would increase.
- Other nodes are affected by the following factor;
  - In the repeated calculation, decrease or increase of evaluation values are spread to other nodes through their outgoing edges.
  - When a closed path is built by merging components, components on the closed path give a part of their evaluation value to other components on the closed path, so evaluation values of them increase.

As a result, evaluation value of the component, that is used by several components among the merged ones, decreases. We also think there is a difference about decreasing degree by how the component is used by other component. Components that receive evaluation values from merged components and other components would be low decreasing degree. On the other hand, components that are used locally only by the merged components would be high decreasing degree. So we set a hypothesis that "Components whose evaluation value goes down are tend to be used by similar code fragments in the merged components."

IV. Implementation of the Analysis Tool

We implemented a tool that calculates two component ranks and compares the difference of each component's evaluation value. This tool is implemented by PHP, and Figure 4 is an overview of the system. Our analysis target is a Java software, and we choose a .java file as a component. The following is the analysis procedure:

1. CCFinder [1] analyzes the target and get a result about code clone. We treat two .java files have a clone relation if they have similar code fragments that are longer than 30 tokens.
2. Classycle's analyzer[1] analyzes the target and get use relations. In the analysis of Classycle's analyzer, we get the following as use relations: inheritance of class, declaration of variables, creation of instances,

[1] http://classycle.sourceforge.net/

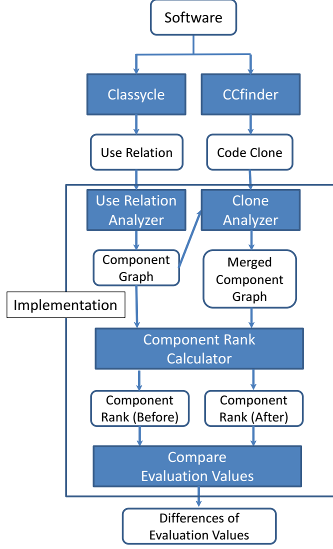

Fig. 4. Overview of the analysis tool

method calls, and reference of fields. Classycle's Analyzer's result is based on relations between classes, so the result between classes are mapped into relations between files, so inner classes and anonymous inner classes and other supplemental classes in a certain file and use relations of them are merged.

3. A component rank before merging is calculated.
4. Our tool merges components that have a clone relation, and use relations are also re-considered on the merged component graph.
5. A component rank after merging is calculated.
6. Two component ranks are compared, and the result of the comparison is outputted as a table.

V. Experiment

A. Objective

By making a comparison between an evaluation value before merging and the one after merging, we performed the following two experiments to confirm that we can detect components that would be a source of code clone generation.

1) We apply our method to source codes of open source projects. We focus attention on not merged components, and not merged ones are categorized as following;

   Group A
   : Components used by several components in the merged components. We can imagine a situation that there are several merged groups. In such case, if the component meets the requirement against at least one of the merged groups, then the component belongs to this.

   Group B
   : Components used by only one of the merged components. This means that component of this category is used only once at the most from all of the merged groups, and some of the merged groups may not use the component.

   Group C
   : Other components. Component of Group C is not used by any of the merged components.

   We confirm that the evaluation values of Group A Components decrease than other components (Group B or C). Through the experiment, we check the adequacy of our consideration, and consider the point which should be taken into consideration in actual use.

2) We get Group A components like experiment1. For each Group A component, we investigate whether merged components use the component in similar way or not, by reviewing code visually, and check the adequacy of a hypothesis that "Components whose evaluation value goes down are tend to be used by similar code fragments in the merged components."

B. Rate of Variability

A number of nodes decreases after merging components. As an example, we suppose the following situation; At first, a number of nodes in the component graph was V_{before}, and an evaluation value of node v_i was $w(v_i)_{before}$. After merging, the number of nodes decreased into V_{after}, and the evaluation value of node v_i became $w(v_i)_{after}$. Because a sum of the evaluation values are always 1, the evaluation value is distributed to each component much relatively. So we define a rate of variability of node v_i as following;

$$\frac{w(v_i)_{after} \times V_{after}}{w(v_i)_{before} \times V_{before}} \times 100 \quad (1)$$

C. Experiment1: About Changes of Evaluation Values

We selected 45 open source projects in SourceForge. This selection was performed via search function of SourceForge, and we selected projects that have both source code files (.java) and bytecode (.jar) and we didn't have other intentions. For each the project, we got one version of the software and applied our method to it, and calculated a rate of variability for each component. We categorized components into Group A, B, and C and calculated an average of the rate for components in Group A, B, and C, respectively. We compared them and confirmed that evaluation value of the components in Group A decreased averagely than the one for Group B and C.

Table I shows an average of the rate for components in Group A per project, the one for Group B and C per project, respectively. Some projects didn't have any components belong to Group A (B, or C), so the number of project for each category was less than 45. We confirmed that an average rate of variability for Group A components decreased 10% and, on the other hand, the one for the components in Group B and C increased slightly. In this way, the average rate of variability for the components in Group A decreased than the one for Group B and C. Group C components were not affected directly from merging nodes, so standard deviation of Group C was smaller than the one of Group A and B.

TABLE I
AN AVERAGE OF RATE OF VARIABILITY FOR EACH PROJECT

	Projects	Average	Standard Deviation	Increased	Decreased
Group A	43	88.5%	19.6	3	40
Group B	41	107.7%	16.2	19	22
Group C	43	104.3%	6.4	20	23

Some projects didn't have any components belong to Group A (B, or C); however, components were present in both Group A and B in 36 projects, and components were present in both Group A and C in 37 projects, respectively. For each project, we calculated the difference between the average rate of variability for the components in Group A and the one for Group B, and also calculated the difference between the one for Group A and the one for Group C. As shown in Table II, the difference between Group B and Group A (B-A) and the one between Group C and Group A (C-A) were the plus value as the 15-20%. In 80-90 percent of projects, we confirmed that Group A components lost evaluation values than other components. Fig. 5 shows a distribution of the differences between Group A and Group B, and Fig. 6 shows a distribution of the ones between Group A and C, respectively. Many projects were plotted in a range of 10% from 20%.

TABLE II
THE DIFFERENCE OF AVERAGE RATE OF VARIABILITY

	Projects	Average	Standard Deviation	projects	
				< 0	> 0
B-A	36	20.8%	20	3	33
C-A	37	15.8%	16.7	5	32

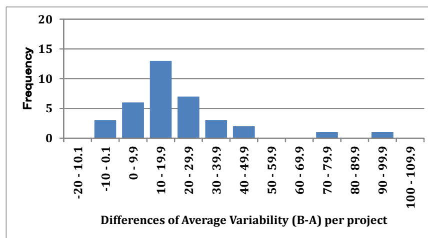

Fig. 5. A Histogram of the Difference of Average Rate (B - A)

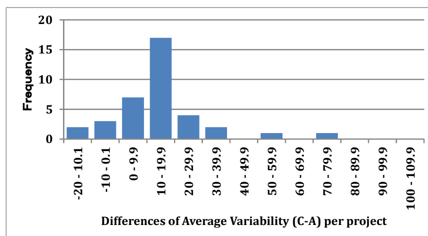

Fig. 6. A Histogram of the Difference of Average Rate (C - A)

We also analyzed these differences statistically by using the Welch's t test, and confirmed that there was a statistically significant difference between Group A and B ($\alpha = 0.01, d = 1.04$, and $1 - \beta = 0.99988$) and there was also a statistically significant difference between Group A and C ($\alpha = 0.01, d = 0.94$, and $1 - \beta = 0.99932$), respectively. From these result, we concluded that the evaluation value of the component that was used by several components of the merged components decreased than other components.

D. Experiment2: Components Used by Merged Components

We selected 34 open source projects from 45 projects in Experiment1 in the interest of time, and not merged components were categorized in the same manner as Experiment1. In this experiment, we focused attention on the Group A components. For each Group A components, we analyzed how the component was used by the merged components by reviewing codes visually, and whether the uses of the components caused producing a code clone or not. This means that the component was used by similar way inside or outside of the detected code clone in the merged components, and their behaviors about using the component were almost the same. We categorized how the component was used in the merged components, and discussed the points should be considered when analyzing code clone with considering use relations.

From the 34 open source projects, we extracted 423 components belong to the Group A, and checked whether the use of the component was a similar way or not. Table III shows the result, and we confirmed that 269 components were used by merged components in a similar way, on the other hand, 154 components were used by merged components in a different way. When it was considered by the percentage, almost two-thirds of Group A components were used by the merged components in a similar way. There were several merged groups, so we confirmed 339 case examples from the 269 components. we classified these 339 case examples based on how the component was used by. In the 195 cases, we confirmed that components were used by a method-call or creation of an instance, and so on, in the similar code fragments. In much of the remaining case, components were inherited by several subclasses, and these subclasses had common descriptions that were recognized as code clone. In other cases, Group A components were exception classes, and these components were used in similar exception handlings.

TABLE III
WAS THE COMPONENT USED BY A SIMILAR WAY?

Components(Group A)	Yes	No	Percentage
423	269	154	64%

Among 34 projects, there were 24 projects that have more than 3 components belong to Group A. For each such project, we calculated a precision ratio for the above-mentioned condition. Fig. 7 shows a distribution of the precision ratio for each project. We confirmed that there was a peak of distribution around the average (64%), on the other hand, there were several projects whose precision ratio are 100%.

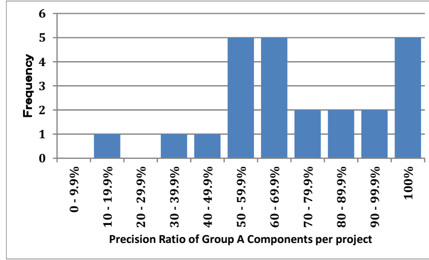

Fig. 7. A Histogram of the Precision for Each Project

We selected 2 projects, dnsjava[2], Jackcess[3] that had more than 20 Group A components. For each project, Group A components were arranged in order of the decreasing rate of the evaluation value, and we calculated a precision ratio for Top X(X=1,2,3,....,20) components, respectively. Fig. 8 is a transition of the precision ratio for top X dnsjava components, and Fig. 9 is a transition of the precision ratio for top X Jackcess components, respectively. In these figures, a solid line shows a precision ratio for Top X components, and a horizontal line shows an average precision ratio of the project. The precision ratios decreased totally with spreading the scope of the ranking, however, the line totally kept over the average precision ratio. For these projects, we considered that components whose evaluation value had decreased a lot didn't always fit into our hypothesis; however, it was often the case that such components fitted into our hypothesis.

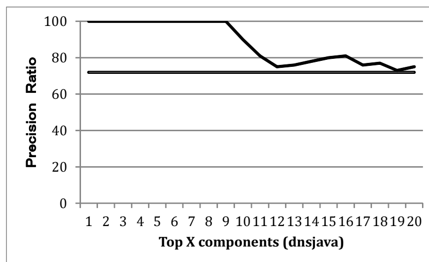

Fig. 8. A Precision Ratio for Top X Components (dnsjava)

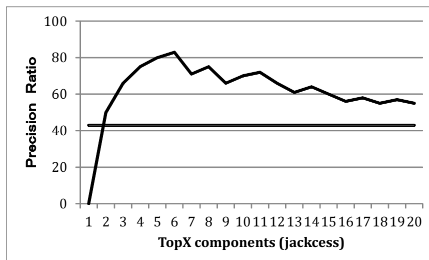

Fig. 9. The Precision Ratio for Top X Components (Jackcess)

[2]http://sourceforge.net/projects/dnsjava/
[3]http://jackcess.sourceforge.net/

VI. Discussion

A. About Result of Experiment1

In the experiment1, we confirmed that the evaluation values of the components, those were used by several components among the merged components (Group A) decreased averagely than the one of others (Group B and C). On the other hand, we also confirmed that evaluation values of some components in Group A increased and the ones of Group B or C components decreased severely. Especially, there were three projects so that the average rate of variability of the Group A components increased, as shown in the Table I. For such cases, we investigated how nodes in the component graph were merged.

- We confirmed that there were several groups in the merged component graph. Some of the Group A component was used by several components in a certain group, however, the component was also used by only one component in another group. In such case, evaluation value of such component tends to increase.
- When a change of the graph structure was too big, or when components whose role was quite different were merged, several closed paths were produced in the merged graph. In such case, the distribution of the evaluation value was strongly affected by the creared closed paths.

From these results, we consider that it isn't desirable to merge components gratuitously, and our method would be improved by setting a restriction for mergence of components based on relativeness between components. For example, we consider that we can extract clone relations that have strong connectivity, by considering relations between packages to which components belong, or by extracting only a larger size of code-clones. As future works, we would like to inspect the effect of such improvement methods by also considering confirgation choice problem [4].

B. About Result of Experiment2

In the experiment2, we confirmed that almost two-thirds of the Group A components were used by the merged components in a similar way. For actual use, we consider the ratio was insufficient to realize the method by simply extracting components that were commonly used by code-clone related components. We consider that sorting components based on a rate of variability seems to be efficient as shown in Fig. 8 and Fig. 9, and a hypothesis that "Components whose evaluation value goes down are tend to be used by similar code fragments in the merged components." worked to some extent. In conjunction with improvement method in Experiment1, we would like to improve our method.

We also classified 339 case examples based on how the component was used by merged components. From a viewpoint of clone analysis based on the use relation, this kind of classification would be useful for identifying a particular kind of use relations are closely related with producing code clones. For example, we consider that the following support would be realizable; When developers introducing some classes that inherit a particular class, it can support by giving advices as

there are already some good examples that inherit the class, and these classes serve as a useful reference. Based on such approach, we can control that the existing code clone derives and increases in various forms in the software. Since such differentiated code clones make it more difficult to detect code clones, so our method would be useful for improvement of the quality of software.

C. Related works

From a viewpoint of use relation (dependency) analysis, architecture recovery is active research area. Zhang represents OO system by using Weighted Directed Class Graph, and proposed a clustering algorithm for recovering high-level software architecture [5]. Constantinou represents hierarchical relationships between components as D-layer, by contracting closed paths in component graph, and investigated relationships between architecture layer and design metrics [6]. In the past research, we proposed a metric representing a change of component rank as an impact for a source code update [7]. We paid our attention to the change of the evaluation value of each component also in this work.

From a viewpoint of clone analysis, Mondal analyzed the stability of several kinds of cloned codes, and they reported that Type3 clones, known as gapped clones, have higher stability than other clones [8]. Antoniol analyzed the evolution of code duplications in 19 versions of the Linux kernel [9]. Yoshida et al. proposed an approach to supporting clone refactoring based on code dependency (e.g., caller-callee, shared variable) among code clones [10]. Their approach presents a coherent set of code clones as a refactoring opportunity. Our research uses component rank model as a basis for consolidating clone information and showing the result, the characteristics of our research is that it pays its attention to the components which the code clone uses in common. In [11], Yamanaka suggested a daily reporting system about modifications related with cloned codes. We consider that changes of component rank may also be fit to be used in such reporting system.

VII. Conclusion

In this paper, we extended component rank model by reflecting relationships of code clones between components on the component graph, and components that have a similar code fragment were merged in the graph. We implemented a system based on the proposed method, and we conducted two evaluation experiments. In the first experiment, we confirmed that the evaluation value of the component, that was used by several components among the merged ones, decreased. In the second experiment, we confirmed that almost two-thirds of the components that were used by the several merged components, were used by merged components in a similar way, and sorting components based on a rate of variability seemed to be efficient. From these results, we could confirm a certain degree of its utility, and a rate of variability seemed to be efficient for filtering not-related components. However, we also think that an improvement of the precision is necessary to sense the utility of our approach in the actual use.

As future works, we would like to inspect the effect of restriction method for mergence of components based on relativeness between components. From a viewpoint of clone analysis based on the use relation, we would like to categorize use relations that caused code clone in more detail, and inspect what kind of use relations tends to generate code clone. After considering those pending problems, we would like to implement a tool where it helps code clones not to increases in various forms. The tool would contribute the support of software-maintenance work and the improvement of the quality of the source code.

Acknowledgment

This research is supported by Nanzan University Pache Research Subsidy I-A-2 for the 2015 academic year. I would like to gratefully and sincerely thank to E.Senga. Original of this research is based on his master thesis written in Japanese [12]. We were supervisors of it and added considerations, experiments and evaluations thoroughly for this paper.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code," *IEEE Transactions. Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[2] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[3] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[4] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 455–465.

[5] Q. Zhangs, D. Qiu, Q. Tian, and L. Sun, "Object-oriented software architecture recovery using a new hybrid clustering algorithm," in *Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, 2010, pp. 2546–2550.

[6] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Towards open source software system architecture recovery using design metrics," in *Proceedings of the 15th Panhellenic Conference on Informatics*, 2011, pp. 166–170.

[7] R. Yokomori, M. Noro, and K. Inoue, "Evaluation of source code updates in software development based on component rank," in *Proceedings of 13th Asia Pacific Software Engineering Conference*, Bangalore, India, 2006, pp. 327–334.

[8] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proceedings of the 27th ACM Symposium on Applied Computing*, 2012, pp. 1227–1234.

[9] G. Antoniol, U. Villano, E. Merio, and M. D. Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[10] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005, pp. 16:1–16:10.

[11] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Industrial application of clone change management system," in *Proceedings of the 6th International Workshop of Software Clones*, 2012, pp. 67–71.

[12] E. Senga, "Evaluation of software components considered code clone," Master's thesis, Dept. of Mathematical and Information Science, Nanzan University, 2013, (In Japanese).