2013 Sixth International Conference on Advanced Computational Intelligence
October 19-21, 2013, Hangzhou, China

Measuring Software Similarity based on Structure and Property of Class Diagram

D. H. Qiu, H. Li, and J. L. Sun

***Abstract*—The measure of software similarity plays a crucial role in software homology detection, software birthmark verification, software watermark, and anti-reverse software engineering, which is vitally important for intellectual property protection of software. However, it is very difficult to measure the similarity between software accurately as it has complex structure and various properties. In this paper, an iterative update process is presented to measure the similarity between the nodes and edges of the class diagrams, in which the structural similarity and the property similarity of the nodes and edges are integrated. A score of software similarity is then derived from the maximum weight matching of the class diagrams. The measure of software similarity is applied to the detection of software homology. An experiment shows that the approach can measure software similarity effectively. The approach is expected to yield satisfactory performance also for other applications in software engineering.**

I. Introduction

THE software engineering field has integrated with the computational intelligence field for a long time [1]. The technologies developed in computational intelligence can be applied to software engineering to address the ever-increasing complexity and size of software systems, to construct models of software processes in the context of imperfect information, and so on. Intellectual property protection for software has become a crucial issue in software engineering, which is important not only for the software industry, but also for other businesses. There is an urgent need to investigate effective approach to protect intelligence property for software, which has been one of the foci of the recent developments in the field of software engineering.

Software homology detection, software birthmark verification, software watermark, and anti-reverse software engineering are useful technologies to protect intelligence property of software. A common problem among them is that measuring similarity between software accurately is not easy. The computational intelligence technologies, such as fuzzy logic, rough set and artificial neural networks have been used to measure software similarity for detecting software homology and software plagiarism [2], [3]. However, this problem still has not been solved well. The main reason is that software is usually with complex structure and large variety of properties. Additionally, the size of software is ever increasing. What is worse, the unauthorized software is usually disguised by semantics-preserving transformations intentionally to avoid punishment. The disguises make the unauthorized software look different and its comprehension difficult. All of these factors lead to an increased difficulty of measuring software similarity.

There have been varieties of approaches for calculating software similarity. In general, these approaches can be divided into four main categories: metric-based approach [4], [5], string-based approach [6]-[13], tree-based approach [14] and graph-based approach [15]-[20]. The metric-based approach extracts some metrics from software and measures software similarity by calculating the distance between the metric vectors. The string-based approach represents software as a sequence of strings or tokens. The software similarity is then calculated through the techniques of sequence analysis. In the tree-based approach, software is first parsed into an Abstract Syntax Tree (AST) with variable names and literal values discarded. Similar subtrees are then searched in AST and the similarity is calculated based on the subtree. The graph-based approach measures the software similarity using the topological structure of software. Usually, the metric-based approach and the sequence-based approach do not consider software structure when calculating software similarity. In contrast, the tree-based approach and the graph-based approach usually focus on the structure only and neglect software properties. As software is structured and has variety of properties, it is certainly meaningful to use both kinds of information together to measure software similarity.

In this paper, we propose a graph-based approach for measuring software similarity, which takes the advantage of the combination of the structure and the properties of class diagrams of software. An iterative update process is first presented to measure the structural similarity between the nodes and edges of the class diagrams. Then, we extract the properties of the classes and the relationships from the class diagrams and measure their property similarity. The property similarity between the classes and the relationships is incorporated into the iterative update process to obtain the comprehensive similarity of the classes and the relationships. After that, we match the class diagrams using a maximum weight matching algorithm. A score of software similarity is finally derived from the matching of the class diagrams. The score of software similarity is applied to the detection of software homology.

The remainder of this paper is organized as follows. Section 2 surveys the existing approaches for measuring

Manuscript received on May 26, 2013. This work was supported in part by the National Natural Science Foundation of China (60873031) and the Fundamental Research Funds for the Central Universities (2011TS145).

D. H. Qiu, H. Li, and J. L. Sun are with the School of Software Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China (Corresponding author: D. H. Qiu; phone: 86-27-87793051; fax: 86-27-87792251; e-mail: qiudehong@163.com).

software similarity. Section 3 describes the motivation and our approach in detail. Section 4 presents the experimental results of measuring software similarity in software homology detection. Section 5 concludes this paper.

II. Work on Measuring Software Similarity

We introduce related work on measuring software similarity. There are four main types of approaches for measuring program similarity: metric-based approach, string-based approach, tree-based approach, and graph-based approach.

The metric-based approach calculates software similarity using some metrics extracted from software. For example, Mayrand et al. extracted the metrics including the number of lines of source, function calls, and edges in Control Flow Graph (CFG) to measure software similarity [4]. Tamada et al. used four characteristics of software source codes in Java, that is, the constant values in field variables, the sequence of method calls, the inheritance structure, and the used classes, to measure software similarity [5]. The metric-based approach is relatively simple. However, as the set of metrics is usually incomplete, the similarity obtained by this approach is not accurate enough.

The string-based approach compares the strings that represent software with each other to measure software similarity. The string-based approach can be further divided into text string-based approach, token string-based approach and bytecode string-based approach. In the text string-based approach, software source code is represented as a sequence of characters. The software similarity is calculated by comparing the character sequences. The text string-based approach is fragile to identifier renaming. In the token-based similarity calculation, software source codes are tokenized and represented as token sequences. The software similarity is measured through finding the duplicated subsequences of the token sequences [6]-[8]. The token-based approach is robust against identifier renaming, formatting and spacing. However, it is fragile to statement reordering, junk code insertion, and control replacement, as these transformations can change the token sequences. In the bytecode string-based approach, the software similarity is calculated using a set of k-length subsequences of the bytecode instructions. The set of k-length subsequences is obtained by sliding a window of length k over the bytecode instructions. Myles and Collberg used the sequences of k contiguous opcodes to measure the similarity to detect software plagiarism [9]. Lim et al. used the sequences of contiguous opcodes partitioned based on their operand stack depth to measure software similarity [10]. The window length influences the similarity calculated by the bytecode string-based approach. In addition, the approach is fragile to structure transformations, as it considers the physical orders of the bytecode instructions only. Besides the main three typical sequences, the system call sequence [11] and the API call sequence [12]-[13] during execution are also used to measure software similarity through the techniques of sequence analysis.

In the tree-based approach, software is first parsed into an AST with variable names and literal values discarded. Similar subtrees are then searched in the AST and the software similarity is calculated based on the common subtrees [14]. The AST tree-based approach is fragile to the disguise techniques such as statement reordering and control replacement.

The graph-based approach can overcome the limitation of the AST tree-based approach. The graph-based approach measures software similarity using the topological structure of software. The software similarity has usually been calculated on Program Dependency Graphs (PDG), CFG, Whole Program Paths (WPP), or System Call Dependence Graph (SCDG). A PDG is a graphic representation of the data and the control dependencies within software [15]. Komondoor and Horwitz measured software similarity through finding isomorphic subgraph of PDG [16]. Liu et al. measured software similarity by mining PDG to discover software plagiarism [17]. A CFG-based approach was proposed by Lim et al. in [18], which calculated software program similarity using k contiguous opcodes in the control flow paths. A WPP-based approach was proposed in [19] to measure software similarity on WPP. Wang et al. measured software similarity based on SCDG that represents the behaviors of software [20].

The technologies developed in computational intelligence, such as fuzzy logic, rough sets and artificial neural networks, have been applied to measuring software similarity. For example, Zhang et al. measured the similarity between software programs based on rough ontology [2]. Fuzzy Logic was used to determine the probabilistic of software plagiarism in [3]. An approach using artificial neural networks was proposed in [21] to match program code.

In the present work, we measure software similarity by using the topological structure and the properties of the class diagram in combination. An iterative update process is presented to measure the similarity between the nodes and edges of the class diagrams. The score of software similarity is finally calculated based on the matching of the class diagrams.

III. Class Diagram-based Approach for Measuring Software Similarity

In this section, we first introduce the class diagram that expresses the structure and the properties of software. Then, we define the structure similarity and the property similarity of the nodes and edges of the class diagrams. Both kinds of similarities are combined and updated in an iterative update process. Subsequently, we describe how to calculate the score of software similarity based on the maximum matching of the class diagrams.

A. Building Weighted Directed Class Diagram

In software engineering, a class diagram is a type of static structure diagram that describes the structure of software by showing the software's classes, the attributes and operations

of the classes, and the relationships among the classes. We abstract class diagram as a weighted directed graph, whose nodes represent classes and edges represent the relationships between the classes. A class encapsulates its attributes and operations together. There is an edge between two classes if one of them has access to a method or data of the other. The classes are extracted from software source codes and weights are assigned to the edges according to the coupling between two classes. According to the coupling framework proposed in [22], we consider three types of coupling between classes, that is, Inheritance Coupling (IC), Method Coupling (MC) and Data Coupling (DC) when assign weight to edges. The weighted directed graph is therefore a hybrid view that reflects not only the structure, but also the properties of software.

Fig.1 shows a part of the class diagram of software Jedit2.3 extracted by the tool of structure101 [23]. Various types of relationships between classes are represented by different types of lines in Fig.1. The dashed line indicates a dependency relationship between two classes. A generalization relationship is shown as a solid line from the more specific class to the more general class, with a hollow triangle at the end of the solid line connected to the more general class. A whole-part aggregation relationship is shown using a solid line with a hollow diamond attached to the class that represents the whole. Structure101 can also output the adjacency matrix of the class diagram. It is more concise than a graph, and especially convenient for the analysis of class diagram.

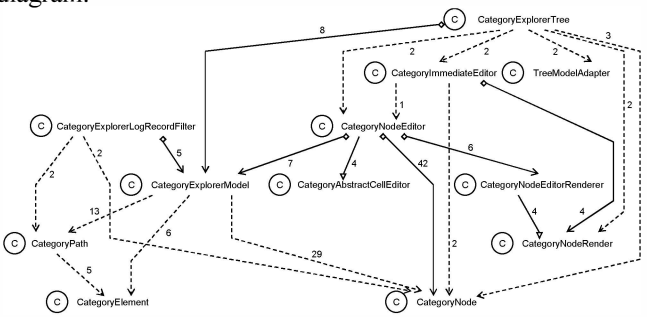

Fig. 1. An example of weighted directed class graph.

B. Structural Similarity of Nodes and Edges

One intrinsic characteristic of software is that it is structured. Class diagram describes the static structure of software. As seen from Fig. 1, class diagram describes the types of classes in software and various kinds of relationships that exist between them. There are three principal kinds of relationships: association, generalization and aggregation. These relationships are all valuable in reflecting inheritance, composition, usage, and connection respectively. The structural information is so important that it should be considered while measuring software similarity.

It can be observed from class diagram that the nodes and edges are closely related. Thus, the node similarity and the edge similarity should not be measured independently. On the contrary, we should consider the edge structural information while measuring node similarity, and vice versa. Given two class diagrams, $G_A(V_A, E_A)$ and $G_B(V_B, E_B)$, that are extracted from two different software respectively, we assume that the structural similarity between the node $v_i \in V_A$ and the node $v_j \in V_B$ is higher if the edges connected to them are more similar; Similarly, the structural similarity between the edge $e_i \in E_A$ and the edge $e_j \in E_B$ is higher if the nodes they link are more similar. That is, the pairwise structural similarity between nodes or between edges within two different class diagrams is higher if their neighborhoods are more similar. We use the iterative method proposed in [24] to compute the structural similarity between the nodes and edges in two class diagrams. As the structural similarity calculated by the iterative method introduces a coupling between the structural information of the edges and nodes, it is appropriate to measure the structural similarity between class diagrams as their nodes and edges are coupled.

The update process proposed in [24] to recursively compute the structural similarity of the nodes and edges is briefly described as below. Let $s_A(i)$ denote the node from which the edge e_i in $G_A(V_A, E_A)$ deviates, and let $t_A(i)$ denote the node at which the edge e_i in $G_A(V_A, E_A)$ terminates. The node set cardinality is $|V_A| = n_A$ and the edge set cardinality is $|E_A| = m_A$. The adjacency structure of $G_A(V_A, E_A)$ can be represented by a pair of $n_A \times m_A$ matrices, A_S and A_T, which are defined as follows.

$$[A_S]_{ij} = \begin{cases} 1 & s_A(j) = i \\ 0 & otherwise \end{cases} \quad (1)$$

$$[A_T]_{ij} = \begin{cases} 1 & t_A(j) = i \\ 0 & otherwise \end{cases} \quad (2)$$

Based on A_S and A_T of the class diagram $G_A(V_A, E_A)$ and B_S and B_T of the class diagram $G_B(V_B, E_B)$, the iterative update process for computing the structural similarity of the nodes and edges is described as below:

$$X_k \leftarrow B_S Y_{k-1} A_S^{\mathrm{T}} + B_T Y_{k-1} A_T^{\mathrm{T}} \quad (3)$$

$$Y_k \leftarrow B_S^{\mathrm{T}} X_{k-1} A_S + B_T^{\mathrm{T}} X_{k-1} A_T \quad (4)$$

where, X_k is a $n_B \times n_A$ matrix, whose element x_{ij} denotes the structural similarity between the node v_i in $G_B(V_B, E_B)$ and the node v_j in $G_A(V_A, E_A)$. Y_k is a $m_B \times m_A$ matrix, whose element y_{pq} denotes the structural similarity between the edge e_p in $G_B(V_B, E_B)$ and the edge e_q in $G_A(V_A, E_A)$. The $\leftarrow$ notation indicates the normalization operation at each step by the Frobenius norm.

If the columns of the matrices X_k and Y_k are stacked into the vectors x_k and y_k respectively, the update process can be reformed as:

$$x_k \leftarrow (A_S \otimes B_S + A_T \otimes B_T) y_{k-1} \equiv G^{\mathrm{T}} y_{k-1} \quad (5)$$

$$y_k \leftarrow (A_S^{\mathrm{T}} \otimes B_S^{\mathrm{T}} + A_T^{\mathrm{T}} \otimes B_T^{\mathrm{T}}) x_{k-1} \equiv G x_{k-1} \quad (6)$$

where $\otimes$ denotes the Kronecker product of matrices. Concatenating the vectors x_k and y_k yields a single matrix update equation:

$$s_k \equiv \begin{bmatrix} x \\ y \end{bmatrix}_k \leftarrow \begin{bmatrix} 0 & G^T \\ G & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{k-1} \equiv M s_{k-1} \quad (7)$$

It has been improved in [24] that, with the initial condition x_0 chosen arbitrarily and with $y_0 = \alpha G x_0$, where α is any positive constant, the iteration of (7) will converge to a unique set x of nonnegative similarity scores between every pair of nodes in $G_A(V_A, E_A)$ and $G_B(V_B, E_B)$ and a corresponding unique set y of nonnegative similarity scores between every pair of edges.

C. Incorporating Property Similarity of Nodes and Edges

The structural similarity of the nodes and edges calculated in the preceding subsection has used only the structural information of the class diagrams. There exists other information about the properties of the classes and the relationships in the class diagrams. A class has three standard compartments, that is, the class name, a list of attributes, and a list of operations. The relationships between classes include association, dependencies, aggregation, and inheritance. A relationship is labeled with a name. Furthermore, a relationship has two ends that may include the information such as rolename, multiplicity, aggregation or composition symbol, and qualifier to express more detail about how the class relates to the other class in the relationships. The properties of the classes and relationships are indispensable to measure software similarity. It is also believed that incorporating the property similarity of the nodes and edges into the iterative update process could improve the accuracy of software similarity.

To calculate the property similarity of classes and relationships, we use the similarity primitives proposed in [25]. Through selecting the appropriate similarity primitives and assigning right weights to them, we define the property similarity of the classes and relationships. A class in class diagram has its name, a set of attributes, and a set of operations. As the class name is easily changed and a little modification of the class name may result in mismatch, we do not include the class name while measuring the property similarity between classes. Thus, for two classes C_1 and C_2, the property similarity between them is defined as:

$$\mathrm{s}(C_1, C_2) = w_a \cdot s_a(C_1(a), C_2(a)) + w_o \cdot s_o(C_1(o), C_2(o)) \quad (8)$$

where, $s_a(C_1(a), C_2(a))$ is the similarity between C_1 and C_2 calculated from their attributes. $s_o(C_1(o), C_2(o))$ is the similarity between C_1 and C_2 calculated from their operations. w_a and w_o are the corresponding weights.

Each attribute of a class is defined by means of a name, a type of data, and visibility. Here, we calculate the similarity between two attributes using the information of data type and visibility, neglecting attribute name also. The data types of two attributes either match or not. This yields the following equation for the data types of the attributes A_1 and A_2.

$$s(A_1(d), A_2(d)) = \begin{cases} 1 & if\ A_1(d) = A_2(d) \\ 0 & otherwise \end{cases} \quad (9)$$

Similarly, the visibilities of two attributes are either identical or different. The following equation is yielded for the visibilities of the attributes A_1 and A_2.

$$s(A_1(v), A_2(v)) = \begin{cases} 1 & if\ A_1(v) = A_2(v) \\ 0 & otherwise \end{cases} \quad (10)$$

Let the attribute set of the classes C_1 and C_2 be A_{c_1} and A_{c_2} respectively, the similarity $s_a(C_1(a), C_2(a))$ between the two attribute sets of C_1 and C_2 is defined as the weighted sum of their data type similarity and visibility similarity.

$$s_a(C_1(a), C_2(a)) = w_d \frac{\sum_{A_i \in C_1, A_j \in C_2} s\left(A_i(d), A_j(d)\right)}{\max(|A_{c_1}|, |A_{c_2}|)} + w_v \frac{\sum_{A_i \in C_1, A_j \in C_2} s\left(A_i(v), A_j(v)\right)}{\max(|A_{c_1}|, |A_{c_2}|)}$$
$$= w_d \cdot s\left(A_{c_1}(d), A_{c_2}(d)\right) + w_v \cdot s\left(A_{c_1}(v), A_{c_2}(v)\right) \quad (11)$$

Similarly, the similarity $s_o(C_1(o), C_2(o))$ between the two operation sets O_{c_1} and O_{c_2} of the classes C_1 and C_2 can be calculated in the same way. An operation is determined by its name, return type, parameters and visibility. Neglecting the operation name, we obtain:

$$s_o(C_1(o), C_2(o)) = w_r \cdot s\left(O_{c_1}(r), O_{c_2}(r)\right) + w_p \cdot s\left(O_{c_1}(p), O_{c_2}(p)\right) + w_v \cdot s\left(O_{c_1}(v), O_{c_2}(v)\right) \quad (12)$$

While measuring the property similarity between edges, we take into consideration three typical relationships between classes: Inheritance Coupling (IC), Method Coupling (MC) and Data Coupling (DC). Inheritance Coupling indicates class C_1 inherits C_2, that is,

$$IC(C_1, C_2) = \begin{cases} 1 & if\ \text{class } C_1 \text{ inherits } C_2 \\ 0 & otherwise \end{cases} \quad (13)$$

Method Coupling indicates class C_1 calls methods of C_2.

$MC(C_1, C_2)$ = the numbers of methods of C_2 called by C_1 (14)

Data Coupling indicates class C_1 uses public data of C_2.

$DC(C_1, C_2)$ = the numbers of data of C_2 used by C_1 (15)

The weight of an edge is the sum of its IC, MC and DC.

$$WR(C_1, C_2) = IC(C_1, C_2) + MC(C_1, C_2) + DC(C_1, C_2) \quad (16)$$

Using the weight of edges, we define the property similarity between two edges, which represents two relationships R_1 and R_2, as below:

$$\mathrm{s}(R_1, R_2) = 1 - \frac{|WR_1 - WR_2|}{\max(WR_1, WR_2)} \quad (17)$$

Let $\mathrm{S}(C_{G_B}, C_{G_A}) = \left[\mathrm{s}(C_i, C_j)\right]_{n_B \times n_A}$ be the property similarity matrix of the classes, whose element $\mathrm{s}(C_i, C_j)$ denote the property similarity between the class C_i in $G_B(V_B, E_B)$ and the class C_j in $G_A(V_A, E_A)$. Similarly, let $\mathrm{S}(R_{G_B}, R_{G_A}) = \left[\mathrm{s}(R_i, R_j)\right]_{m_B \times m_A}$ be the property similarity matrix of the relationships, whose element $\mathrm{s}(R_i, R_j)$ denote the property similarity between the relationship R_i in

$G_B(V_B, E_B)$ and the relationship R_j in $G_A(V_A, E_A)$. The property similarity between the classes and relationships can be incorporated into the iterative update process through the equations below:

$$X_k \leftarrow X_k + \beta \cdot \mathrm{S}\left(C_{G_B}, C_{G_A}\right) \quad (18)$$

$$Y_k \leftarrow Y_k + \gamma \cdot \mathrm{S}\left(R_{G_B}, R_{G_A}\right) \quad (19)$$

where, β and γ are the incorporation parameters.

D. Calculating the Score of Software Similarity

We calculate the score of software similarity based on the matching of the class diagrams. Here, we use a linear programming approach to compute a bipartite matching with maximum similarity between two sets of nodes, which is briefly described as below.

Given two class diagrams $G_A(V_A, E_A)$ and $G_B(V_B, E_B)$ extracted from software respectively, the matching between them is described by a 0/1-idicator vector $\mathrm{x} \in \{0,1\}^{n_B \times n_A}$. The element $\mathrm{x_{ji}} = 1$ indicates the node $\mathrm{v_i} \in \mathrm{V_A}$ is matched to the node $\mathrm{v_j} \in \mathrm{V_B}$. Otherwise, $\mathrm{x_{ji}} = 0$. The optimal bipartite matching between $G_A(V_A, E_A)$ and $G_B(V_B, E_B)$ is obtained by solving the linear integer program below:

$$\max_{\mathrm{x}} \mathrm{s^T x} \quad (20)$$

$$\text{s.t. } \mathrm{A}_{n_B}\mathrm{x} = \mathrm{e_K},\ \mathrm{A}_{n_A}\mathrm{x} \le \mathrm{e_L},\ \ \mathrm{x} \in \{0,1\}^{n_B \times n_A}$$

where, $s = \left(s(1,1), \cdots s(i,j), \cdots s(n_B, n_A)\right)^T$ is a vector, whose element $s(i,j)$ is the similarity between the vertices $v_i \in V_\mathrm{A}$ and $v_j \in V_\mathrm{B}$. The matching constraints are defined by the constraint matrices A_{n_B} and A_{n_A}. The matrix $\mathrm{A} = \left(A_{n_B}^\mathrm{T}, A_{n_A}^\mathrm{T}\right)^\mathrm{T}$ composed of A_{n_B} and A_{n_A} is the incidence matrix of the bipartite graph. The score of software similarity is therefore defined as below.

$$\mathrm{s}(G_A, G_B) = \frac{\mathrm{s^T x}}{\max(n_A, n_B)} \quad (21)$$

IV. Experiment and Discussion

In order to evaluate the approach of measuring software similarity, we used it to detect homology in genealogy of evolving software. The software in the same genealogy often contains a large number of similar source code fragments across versions as they share common ancestry. The similar source code fragments are typically introduced when the source codes are inherited from previous versions, or duplicated for programming convenience. The similar source code fragments descended from a common ancestry are called homologous codes. Software homology detection is especially important in the current flourishing software market, as it can prevent the misuse of unauthorized software source codes effectively.

We chose the eMule family as the test dataset. eMule is a free peer-to-peer file sharing software for Microsoft Windows. Since July 2002 eMule has been free software, released under the GNU General Public License. eMule is coded in C++ using the Microsoft Foundation Classes. The distinguishing features of eMule are the direct exchange of sources between client nodes, fast recovery of corrupted downloads, and the use of a credit system to reward frequent uploaders. Furthermore, eMule transmits data in zlib-compressed form to save bandwidth. Its popularity has led to eMule's codebase being widely used as the basis of cross-platform clients. eMule 0.50a updated on 24 May, 2012 is the current new version. We chose six versions of eMule and extracted their class diagrams. Table 1 shows the number of the classes in each version of eMule. The class diagram of the eMule in version 0.44d that contains 378 classes is shown in Fig.2.

TABLE I
THE NUMBER OF CLASSES CONTAINED IN EMULE

0.10a	0.20b	0.25b	0.30a	0.42a	0.44d
36	105	158	229	326	378

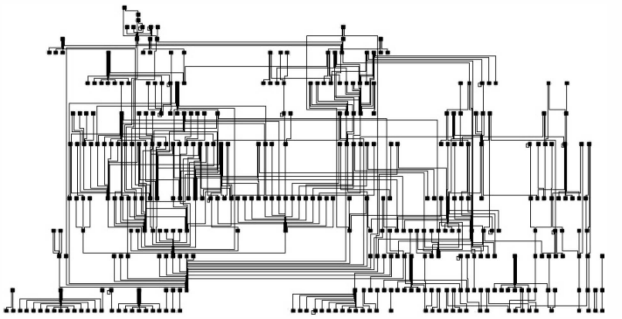

Fig. 2. The class diagram of the eMule in version 0.44d.

Another free open source peer-to-peer software named Ares is selected for comparison. Ares is a successful peer-to-peer application that enables users to share any digital file including images, audio, video, software, documents, etc. The version of Area that we used in comparison experiment is 2.1.2, which contains 132 classes.

Fig.3 shows the similarity between eMule pairs in continuous versions. Meanwhile, Fig.4 shows the similarity between Ares and eMule in different versions. It can be observed that the score of the similarity between eMule pairs in continuous versions is usually larger than 0.5. However, the score of the similarity between Ares and eMule in different versions is lower than 0.31. According to the score of software similarity, we can determine whether the software is homologous or not.

However, extensive experiments should be conducted to test this approach for calculating software similarity more widely. This is one of the main tasks in the future work. Furthermore, the performance of this approach for measuring software similarity depends on many factors. In particular, the weight parameters and the kinds of similarity primitives have a dramatic impact on the final score. We should investigate more on the selection of weight parameters and similarity primitives. Finally, we need to integrate the iterative update process that calculates the similarity score of software into the similarity-based classifying algorithms to complete the specific tasks in software engineering.

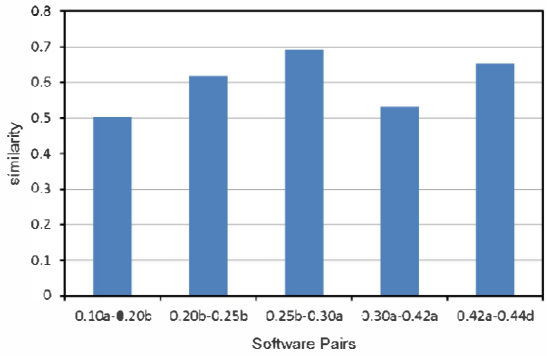

Fig. 3. The similarity between eMule pairs in different versions.

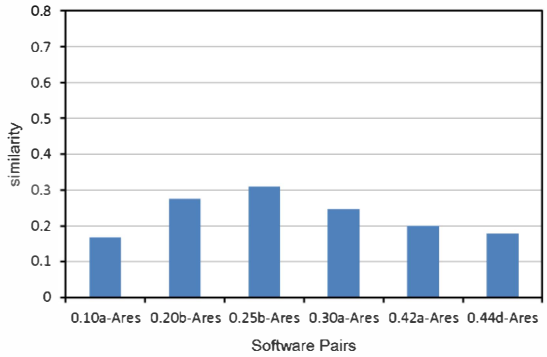

Fig. 4. The similarity between eMule and Ares.

V. CONCLUSION

Measuring software similarity is critical for solving the problems in software engineering such as software homology detection, software birthmark verification, and software clone detection. The approach proposed in this paper takes advantage of the combination of structure similarity and property similarity of class diagrams. Both kinds of similarities are incorporated into an iterative update process to calculate the score of software similarity. Experiments demonstrate that the approach measures software similarity effectively. In future work, we will incorporate the proposed approach into the similarity-based classifier to detect homologous software and compare the proposed approach with the existed approaches such as metric-based, string-based, tree-based and graph-based approaches for measuring software similarity.

REFERENCES

[1] J. Lee, *Software Engineering with Computational Intelligence (Studies in Fuzziness and Soft Computing)*. New York: Springer-Verlag, 2003.

[2] P. Zhang, G. Y. Wang, C. M. Tao, and H. Luo, "Approach to compute program similarity based on rough ontology," *Journal of Chongqing University of Posts and Telecommunications (Natural Science Edition)*, vol. 20, pp. 737–741, June 2008.

[3] D.-K. Chae, S.-W. Kim, J. Ha, S.-C. Lee, and G. Woo, "Software plagiarism detection via the static API call frequency birthmark," in *Proc. 28th Annual ACM Symposium on Applied Computing*, Coimbra, Portugal, 2013, pp. 1639–1643.

[4] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. 12th International Conference on Software Maintenance*, Monterey, CA, USA, 1996, pp. 244–253.

[5] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Java birthmark-detecting the software theft," *IEICE Trans. Inform. Syst.*, vol. E88-D, pp. 2148–2158, Sept. 2005.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 654–670, July 2002.

[7] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. UCS.*, vol. 8, pp. 1016–1068, Nov. 2012.

[8] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software Pract. Exper.*, vol. 37, pp. 151–175, Feb. 2007.

[9] G. M. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. 20th ACM Symposium on Applied Computing*, Santa Fe, New Mexico, USA, 2005, pp. 314–318.

[10] H. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of Java applications via a static birthmark based on weighted stack patterns," *IEICE Trans. Inform. Syst.*, vol. E91-D, pp. 2323–2332, Sept. 2008.

[11] K. Okamoto, H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Dynamic software birthmarks based on API calls," *IEICE Trans. Inform. Syst.*, vol. E89-D, pp. 1751–1763, Aug. 2006.

[12] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA, 2007, pp. 274–283.

[13] H. Park, S. Choi, H. Lim, and T. Han, "Detecting code theft via a static instruction trace birthmark for Java methods," in *Proc. 6th IEEE International Conference on Industrial Informatics*, Daejeon, Korea, 2008, pp. 551–556.

[14] I. D. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. 14th International Conference on Software Maintenance*, Bethesda, Maryland, USA, 1998, pp. 368–377.

[15] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. 8th Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001, pp. 301–309.

[16] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. 8th International Symposium on Static Analysis*, Paris, France, 2001, pp. 40–56.

[17] C. Liu, C. Chen, J. Han, and P.S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Philadelphia, USA, 2006, pp. 872–881.

[18] H. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information" *Software Tech.*, vol. 51, pp. 1338–1350, Sept. 2009.

[19] G. M. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," *Information Security Conference, Lecture Notes in Computer Science*, 3225, pp. 404–415, 2004.

[20] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc. 16th ACM conference on Computer and Communications Security*, Chicago, Illinois, USA, 2009, pp. 280–290.

[21] T. M. Nascimento, D. R. Boccardo, C. B. Prado, R. C. S. Machado, and L. F. R. C. Carmo, "Program matching through code analysis and artificial neural networks," *Int. J. Soft. Eng. Knowl. Eng.*, vol. 22, pp. 225–241, Feb. 2012.

[22] J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented Systems," Technical Report, Univ. of Klagenfurt, 1994. Available: ftp://ftp.ifs.uni-linz.ac.at/pub/publications/1993/0293.ps.gz.

[23] Software architecture management for Java, .Net, C/C++ and anything with strcture101. Available: http://www.headwaysoftware.com/products/structure101/index.php.

[24] L. A. Zager and G. C. Verghese, "Graph similarity algorithms," Applied Mathematics Letters, vol. 21, pp. 86–94, Jan. 2008.

[25] D. J. A. van Opzeeland, C. F. J. Lange, and M. R. V. Chaudron, "Quantitative techniques for the assessment of correspondence between UML designs and implementations," in *Proc. 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Glasgow, United Kingdom, 2005, pp. 1–17.