Requirements Engineering (2020) 25:213–229
https://doi.org/10.1007/s00766-019-00317-w

ORIGINAL ARTICLE

Structural similarity measure between UML class diagrams based on UCG

Zhongchen Yuan[1] · Li Yan[2] · Zongmin Ma[2]

Received: 19 October 2018 / Accepted: 10 June 2019 / Published online: 18 June 2019

Abstract
In software reuse, the reuse of UML class diagram produced in design phase has received more attention due to the important influence on the following developing process. The reuse is based on similarity. The similarity between class diagrams contains semantic and structural aspects. The existing works focus on semantic similarity, while the structural similarity is little paid attention to. The structure of class diagram can be categorized into two aspects: *intra-structure* and *inter-structure*. The intra-structure refers to the composition of each class, and the inter-structure is represented as the relationships between classes. So, the structural similarity measure should be carried out from these two aspects. In this paper, we propose to use a graph named UML class graph (UCG) to represent a class diagram for the structural similarity measure. An algorithm based on UCG Maximum Common Subgraph Sequence is proposed for the inter-structure similarity measure, and UCG edit distance is proposed and introduced to the intra-structure similarity measure. The experimental results show that our proposed approach is effective within a domain or across domains.

Keywords Software reuse · UML class diagram · Structural similarity · Inter-structure · Intra-structure · UCG

1 Introduction

Software reuse can save development costs and time to improve software development process [1]. With the increasing complexity of software, software reuse has been involved in each phase of software life cycle, including design, testing or even maintenance, not just limited to code [2, 3]. Software design has an enormous influence on the following development process [4, 5], so the reuse of software design is promising. Class diagrams produced in design phase can clearly show the static structure of a system by modeling objects and relationships between objects [6]. Currently, the reuse of class diagrams has received more attention [7, 8]. The reuse architecture of class diagrams is shown as Fig. 1.

It is shown in Fig. 1 that the reuse architecture of class diagrams contains four stages. The original class diagrams are retrieved, adjusted and then applied for new projects. The newly developed class diagrams are finally added into the repository for future reuse. Among them, the retrieval that is based on similarity measure is a key. The existing works on similarity measure focus on semantics [9]. However, class diagram contains not only semantics but also structure [10]. Class diagrams for modeling a software system are generally created by a team of developers who may have different experiences and knowledge backgrounds. It is a common case that the created class diagrams are not exactly consistent even for the development of the same project.

Let us look at an example. Suppose that we have a query class diagram shown in Fig. 2a as input. Then, with a semantics-based retrieval, the class diagrams containing Fig. 2a, b should be retrieved in the reuse repository. It can be seen that the retrieved class diagrams may have different structures due to their different developing concerns. Here, Fig. 2a is a student-centered design and Fig. 2b is a lesson-centered design. However, it is possible that only the class diagrams containing Fig. 2a are required in an application, including the related artifacts of these class diagrams. At this point, the class diagrams containing Fig. 2b would not appear in the retrieval list with respect to the structural information of the query class diagram. Let us look at another example.

✉ Zongmin Ma
zongmin_ma@yahoo.com

1 School of Software, Northeastern University, Shenyang 110819, China

2 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

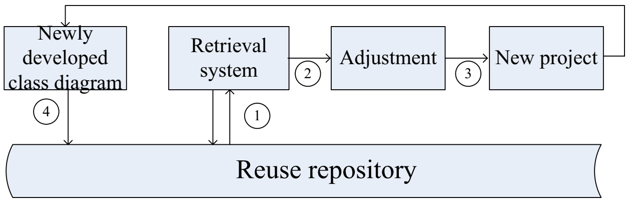


Fig. 1 The reuse architecture of class diagrams

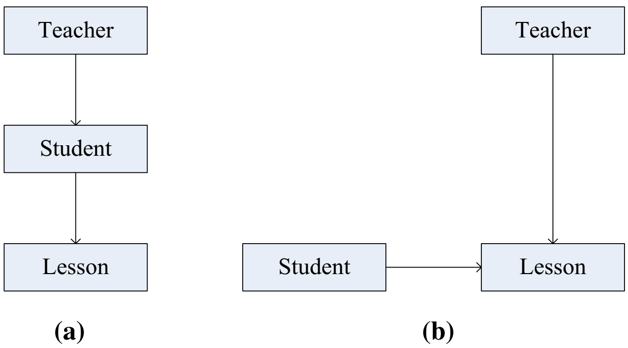


Fig. 2 UML class diagram examples

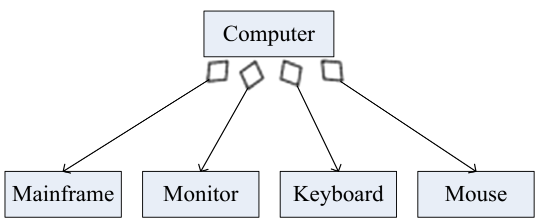


Fig. 3 A class diagram modeling a computer composition

For the query class diagram shown in Fig. 3, which is used to model the composition of a computer, there may not be any class diagrams that model the same project as the query class diagram in the reuse repository. As a result, no class diagrams would be retrieved if a semantics-based retrieval is applied. However, there may be some structurally similar class diagrams from different projects in the reuse repository (e.g., the class diagram modeling a vehicle composition in Fig. 4), which can be applied as a useful reference to construct new related class diagrams. Therefore, in addition to the semantics of class diagrams, the retrieval of class diagrams needs to consider the structures of class diagrams also for structural reuse. The key of structural retrieval for structural reuse is the structural similarity measure.

So far, while more attention has been paid to the semantic similarity measure of class diagrams, little work has been carried for the structural similarity measure of class diagrams. In this paper, we concentrate on the structural similarity measure of class diagrams. For this purpose, we

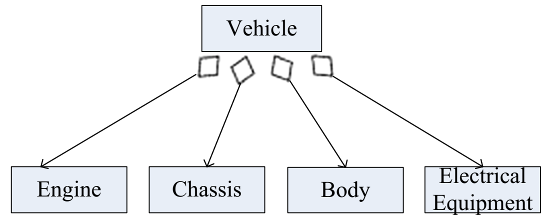


Fig. 4 A class diagram modeling a vehicle composition

propose a graph model named UCG (UML class graph) to represent class diagram. On the basis of the UCG model, we propose the algorithms for the structural similarity measure of class diagrams. The main contributions of this paper are summarized as follows.

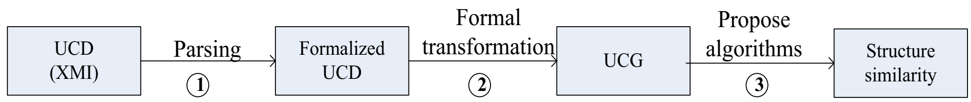
(1) We propose to consider the reuse of class diagrams from a structural perspective.
(2) We propose the structural similarity measure method for the structural reuse, where an UCG is proposed to represent a class diagram, an algorithm based on UMCSS is proposed for the inter-structure similarity measure and UCG edit distance is proposed for the intra-structure similarity measure.
(3) We carry out an experiment to show the effectiveness of the proposed method.

The rest of this paper is organized as follows. The related work is presented in Sect. 2. Section 3 presents the generic procedure of model transformation, formally defining UML class diagram and UML class graph and providing the transformation rules. The structural similarity measure between UML class graphs is proposed in Sect. 4. Section 5 presents an experiment and analyzes the experimental results. Section 6 concludes this paper.

2 Related work

The advance is mainly reflected in semantic similarity since the reuse of software artifacts (e.g., code, component and design model) has been valued [11–20]. The most commonly used approach is that, a reusable artifact is described as a few features, each feature is assigned, and then the similarity between artifacts is calculated using the difference between features [11, 13, 16–18, 20]. The definition and assignment of features is generally a manual process that requires more domain knowledge and searching artifacts for reuse is based on keyword. In [21], a method called case-based reasoning is proposed, in which previous experiences are described as cases (problem and solutions) stored in a case library. Given a query condition, the most similar cases are received and then adapted for reuse in new project. With

Fig. 5 Procedure of using UCG to measure the structural similarity between UCD



the development of Semantic Web, more ontologies (e.g., WordNet) [22] are developed and applied to some fields such as knowledge engineering and information retrieval [23]. Ontology-based similarity measure is proposed [24, 25], in which domain and application ontologies are combined to improve the accuracy of semantic similarity measure [15]. A relationship is usually represented as a vector of end class and type in [15, 19, 20], then the distance between vectors is used to measure the similarity between relationships, which can be essentially viewed as a kind of semantic measure and only applied to the same projects. Certainly, still a few methods have been proposed for the structural similarity measure [19, 26–30]. In [19, 28], the neighborhood information is used to measure the similarity between relationships. A sequence diagram is represented as a conceptual graph for the similarity measure in [29], in which object name corresponds to vertex and message corresponds to edge. Then the matching is based on the labels of vertices and name of edges, which falls into a semantic similarity category. In [30], the state machine diagram is represented as a digraph for the similarity measure and the similarity measure is based on an adjacency matrix representation of different edges. In [27], a model query language is designed to rewrite a class diagram for the structural matching, where a depth-first algorithm is applied for searching the maximum common parts. Note that, when the number of relationships contained in the class diagrams is small, this approach can work well because few common substructures exist among them. As the size of class diagrams increases, the number of common substructures may be more than one and it is inaccurate to use this method for calculating the structural similarity. In addition, the text-based representation is inappropriate to represent class diagram because the structure of class diagram is not represented intuitively. So, a graphical and accurate approach is desirable for the structural similarity measure between class diagrams.

The structure of class diagram can be categorized into two aspects: intra-structure and inter-structure. The intra-structure refers to the composition of each class, and the inter-structure is represented as relationships between classes. Both the intra-structure and inter-structure are all within the scope of consideration in this paper. We apply a graph [29, 30] to represent a class diagram for the structural similarity measure. The vertices and edges of an UCG are classified into different types, and the structural matching is based on the edge tags rather than vertices. An UMCSS-based algorithm is proposed for the inter-structure similarity measure, and UCG edit distance is proposed for the intra-structure similarity measure. The feature vector method [11, 13, 16–18, 20, 24, 25] and the vertex label method [29, 30] pay their attention on the semantics rather than the actual structure. Compared with the semantics-based method, the method proposed in the paper does not care for the semantics (end class) and the matching is just based on the tags of edges. This can be viewed as a structural matching in nature, and it can also be applied to the structural reuse of the same domain and across domains. In [27], a model query language method is proposed. Our method considers more common substructures in addition to the maximum common substructure, and this can improve the accuracy. It is especially true for the similarity measure between class diagrams with a large size. Additionally, the graphical representation of a class diagram's structure is more intuitive than the text representation.

3 Model transformation

OMG (Object Modeling Group) defines standard DTD (Document Type Definition) for UML model file. Then an UML model is described in an XMI (Extended Mark-up Language Interchange) document based on DTD standard [31]. The structural similarity measure between class diagrams can be attributed to *model matching*. There are two strategies to solve the issue of model matching. The first one is to put forward algorithms on the model, and the second one is to transform the model into another model and then put forward algorithms on the new model. Here we chose the latter. A graph called UCG is proposed to represent an UML class diagram (denoted as UCD) for the structural similarity measure in this paper. The procedure is described in Fig. 5.

Obviously, this process consists of three steps. Among them, parsing XMI is to obtain all elements of class diagram. Any XML parser based on SAX (Simple API for XML) can be used to parse XMI model file and then obtain the elements (i.e., class, attribute, operation and relationship) [32]. All these elements obtained by parsing provide a preparation for formalizing class diagram. To transform UCD to UCG, the transformation rules need to be defined and the structural information of UCD must be fully reflected in UCG. On the basis, the structural similarity between UCD is converted to the structural similarity between UCG. Finally, algorithms are proposed for the structural similarity measure.

UCD and UCG are formally defined, and then, the transformation rules from UCD to UCG are summarized in the following subsections.

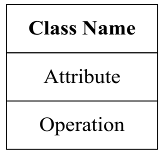


Fig. 6 A class composition

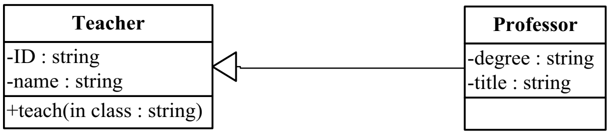


Fig. 7 An example of UML class diagram

3.1 UML class diagram

An UML class diagram is used to model the static structure of a system, which consists of classes and relationships between classes [6]. Being an abstract representation of a set of objects with the same properties, a class shown in Fig. 6 is composed of attributes and operations. A relationship existing between classes is mainly classified into six categories: association, generalization, dependence, aggregation, composite and realization. An example shown in Fig. 7 is a fragment of a class diagram from an education domain. It contains two classes named "Teacher" and "Professor," and one relationship of generalization, indicating class "Professor" inherits from class "Teacher."

Definition 1 We use a 5-tuple to formally define an UML class diagram and have UCD = (C, A, O, P, R).

(1) C is a set of classes, where $C = \{c_1, c_2, c_3, \ldots, c_k\}$ and c_i is a class;
(2) A is a set of attribute sets, where $A = \{A_1, A_2, \ldots, A_k\}$, A_i is a set of attributes contained in class c_i, $A_i = \{a_{i1}, a_{i2}, \ldots, a_{im}\}$, and a_{ij} is the jth attribute of class c_i;
(3) O is a set of operation sets, where $O = \{O_1, O_2, \ldots, O_k\}$, O_i is a set of operations contained in class c_i, $O_i = \{o_{i1}, o_{i2}, o_{i3}, \ldots, o_{in}\}$, and o_{ik} is the kth operation of class c_i;
(4) P is a set of all the parameters, where $P = \{P_1, P_2, \ldots, P_k\}$, P_i is a set of parameters contained in all the operations of class c_i. $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$, P_{ij} is a set of parameters contained in the operation o_{ij}, $P_{ij} = \{p^1_{ij}, p^2_{ij}, p^3_{ij}, \ldots, p^t_{ij}\}$, and p^t_{ij} is the tth parameter of operation o_{ij};
(5) R is a set of relationships, where $R = \{r_{ij} | 1 \leq i, j \leq |C|$ *and* $i \neq j\}$, $r_{ij} = (c_i, t_x, c_j)$ is a relationship between class c_i and c_j, $t_x \in T$ is the type of r_{ij}, and $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ is a set of relationship types. Here t_1, t_2, t_3, t_4, t_5 and t_6 corresponds to association, generalization, aggregation, composition, dependency and realization, respectively.

For the class diagram in Fig. 7, two classes "Teacher" and "Professor" are denoted as c_1 and c_2, respectively; for class "Teacher," attribute "ID" is denoted as a_{11}, attribute "name" is denoted as a_{12}, operation "teach" is denoted as o_{11}, and parameter "class" is denoted as p^1_{11}; similarly, the attributes "degree" and "title" of class "Professor" are denoted as a_{21} and a_{22}, respectively; the generalization relationship between class "Teacher" and "Professor" is then denoted as r_{21}, $r_{21} = (c_2, t_2, c_1)$.

3.2 UML class graph

A graph is an ordered pair (V, E), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and an edge exists between two vertices [33]. As a powerful modeling tool, a graph is applied to a series of fields, ranging from computer network to biomedical science [34]. A core in graph applications is the issue of *model matching* [35]. The structure of an UCD is similar to a graph: Classes of an UCD correspond to vertices of a graph and relationships of an UCD correspond to edges of a graph. So, a graph is chosen to represent an UCD for the structural similarity measure. In this section, we propose an UCG to represent an UCD. Being different from a general digraph, an UCG consists of various types of vertices and edges to correspond to different elements in an UCD.

Definition 2 An UML class graph is defined as UCG = (V, E, L).

(1) V denotes all vertices of an UCG, where $V = CV \cup AV \cup OV \cup PV$.

- CV is a set of class vertices and $CV = \{cv_1, cv_2, \ldots, cv_k\}$, where cv_i is the ith class vertex.
- AV is a set of sets of attribute vertices and $AV = \{AV_1, AV_2, \ldots, AV_k\}$, where $AV_i = \{av_{i1}, av_{i2}, \ldots, av_{im}\}$ is a set of attribute vertices connecting to class vertex cv_i and av_{ij} is the jth attribute vertex.
- OV is a set of sets of operation vertices and $OV = \{OV_1, OV_2, \ldots, OV_k\}$, where $OV_i = \{ov_{i1}, ov_{i2}, \ldots, ov_{in}\}$ is a set of operation vertices connecting to class vertex cv_i and ov_{ij} is the jth operation vertex.
- PV is a set of all parameter vertices and $PV = \{PV_1, PV_2, \ldots, PV_k\}$, where $PV_i = \{PV_{i1}, PV_{i2}, \ldots, PV_{in}\}$ is a set of parameter vertices connecting to all operation vertices that are connected to class vertex cv_i, $PV_{ij} = \{pv^1_{ij}, pv^2_{ij}, \ldots, pv^f_{ij}\}$ is a set of parameter vertices connecting to the operation vertex ov_{ij}, and pv^t_{ij} is the tth parameter vertex.

(2) E denotes all edges of an UCG, where $E = AE \cup OE \cup PE \cup RE$.

Table 1 Element tags of UCG

No.	Element type		Tag
1	Vertex	Class vertex	v_0
2		Attribute vertex	v_1
3		Operation vertex	v_2
4		Parameter vertex	v_3
5	Edge	Attribute edge	e_a
6		Operation edge	e_o
7		Parameter edge	e_p
8		Association edge	e_1
9		Generalization edge	e_2
10		Aggregation edge	e_3
11		Composition edge	e_4
12		Dependency edge	e_5
13		Realization edge	e_6

- $AE \subseteq CV \times AV$ is a set of attribute edge sets and $AE = \{AE_1, AE_2, \ldots, AE_k\}$, where $AE_i = \{ae_{i1}, ae_{i2}, \ldots, ae_{im}\}$ denotes a set of attribute edges connecting class vertex cv_i and $ae_{ij} = (cv_i, av_{ij})$ is an attribute edge from cv_i to av_{ij}.
- $OE \subseteq CV \times OV$ is a set of operation edge sets and $OE = \{OE_1, OE_2, \ldots, OE_k\}$, where $OE_i = \{oe_{i1}, oe_{i2}, \ldots, oe_{in}\}$ denotes a set of operation edges connecting class vertex cv_i and $oe_{ij} = (cv_i, ov_{ij})$ is an operation edge from cv_i to ov_{ij}.
- $PE \subseteq OV \times PV$ is a set of parameter edges and $PE = \{PE_1, PE_2, \ldots, PE_k\}$, where $PE_i = \{PE_{i1}, PE_{i2}, \ldots, PE_{in}\}$, $PE_{ij} = \{pe^1_{ij}, pe^2_{ij,\ldots}, pe^f_{ij}\}$, and $pe^t_{ij} = (ov_{ij}, pv^k_{ij})$ is a parameter edge from ov_{ij} to pv^k_{ij}.
- $RE \subseteq CV \times CV$ is a set of relationship edges and $RE = \{re_{ij} | 1 \le i, j \le |CV| \text{ and } i \ne j\}$, where $re_{ij} = (cv_i, e_x, cv_j)$ is a relationship edge from cv_i to cv_j, $e_x \in ET$ is a tag of re_{ij} and $ET = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ is a set of relationship edge tags.

(3) L is a label function, which denotes the label of a vertex, $L = L^C + L^A + L^O + L^P$. $L^C(cv_i)$, $L^A(av_{ij})$, $L^O(ov_{ij})$ and $L^P(pv^k_{ij})$ denote the label of class vertex cv_i, attribute vertex av_{ij}, operation vertex ov_{ij} and parameter vertex pv^k_{ij}, respectively.

In a general digraph, the differences among vertices are based on labels and all edges are seen to be identical except for different weights. The vertices and edges of an UCG, however, are identified as different types (as mentioned above). Each type of elements plays a different role in an object that is composed of several different types of elements. These different types of vertices and edges are denoted as different tags in Table 1 to distinguish each other.

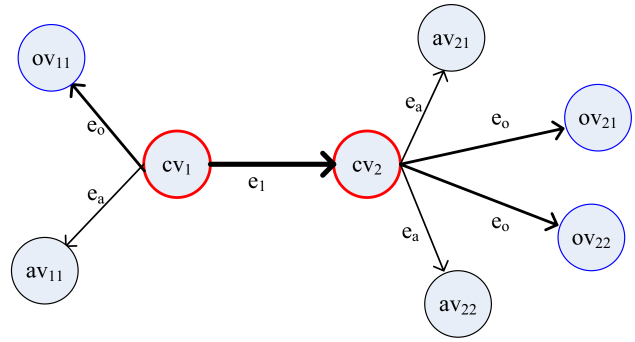


Fig. 8 An UCG application case

In the real world, these elements that make up an object are usually multiple types instead of single type, so the modeling tools like UCG have a wide range of applications. Let us look at an application example of UCG in network topology design. In Fig. 8, a higher bandwidth is designed between two key nodes as the backbone, say e_1, and a relatively low bandwidth is assigned between a key node and a general node, say e_a and e_o, shown. A class vertex is a key node, and an attribute vertex and an operation vertex are considered as general nodes, which are different from each other and marked with different colors. In addition, different bandwidths are denoted as edges with different pounds. The same idea can be applied to highway construction planning, where higher-quality roads should be built between key cities and the standards among other cities are less demanding.

3.3 Transformation rules

Transformation rules from UCD to UCG are proposed in this section. Here the UCG is applied for measuring the structural similarity instead of a complete matching. So, we do not consider the multiplicity of relationship here. The related permissions (e.g., public, private, and protected) of attribute and operation are also ignored in this paper. In the following, we present the detailed transformation rules.

- **Rule 1: class → class vertex**
  Class c_i in an UCD is transformed into a class vertex cv_i in an UCG and the name of class c_i becomes the label $L^C(cv_i)$ of cv_i.
- **Rule 2: attribute → attribute vertex and attribute edge**
  Attribute a_{ij} of class c_i in an UCD is transformed to an attribute vertex av_{ij} in an UCG and the name of a_{ij} becomes the label $L^A(av_{ij})$ of av_{ij}. Then an attribute edge ae_{ij} between cv_i and av_{ij} is created and the direction is from cv_i to av_{ij}. The type of attribute a_{ij} is assigned to the tag e_a of attribute edge with a mark (e.g., $ta_1, ta_2, \ldots, ta_n$).

- **Rule 3: operation (parameter) → operation vertex and operation edge (parameter vertex and parameter edge)**
  Operation o_{ij} of class c_i in an UCD is transformed to an operation vertex ov_{ij} in an UCG. Then an operation edge oe_{ij} between cv_i and ov_{ij} is created and the direction is from cv_i to ov_{ij}. The name of o_{ij} becomes the label $L^O(ov_{ij})$ of the operation vertex ov_{ij} and the return type of operation o_{ij} is assigned to the tag e_o of operation edge oe_{ij} with a mark (e.g., rt_1, rt_2, …, rt_n). Being different from an attribute, an operation may contain some parameters. A parameter is defined by both name and type. A parameter can be handled in a similar way as an attribute, but a parameter edge is created between operation vertex and parameter vertex. So, parameter p^t_{ij} in an UCD is transformed into a parameter vertex pv^t_{ij} in an UCG. Then a parameter edge pe^t_{ij} between pv^t_{ij} and ov_{ij}, is created and the direction is from ov_{ij} to pv^t_{ij} The name of parameter p^t_{ij} becomes the label $L^P(pv^t_{ij})$ of parameter vertex pv^t_{ij} and the type of parameter p^t_{ij} is assigned to the tag e_p of parameter edge pe^t_{ij} with a mark (e.g., tp_1, tp_2, …, tp_n).
- **Rule 4: relationship → relationship edge**
  Relationship r_{ij} between class c_i and c_j in an UCD is transformed into a relationship edge re_{ij} between class vertex cv_i and cv_j in an UCG. Regarding the direction and tags of relationship edge, Fig. 9 presents the details.

With the transformation rules, the UCD in Fig. 7 is converted into an UCG in Fig. 10. Here different types of vertices are denoted with different colors for distinguishing each other.

All the elements from an UCD can be transformed into corresponding vertices and edges of an UCG based on the above transformation rules. The structure of an UCD is represented as the structure of an UCG. The following is a summary of the model transformation.

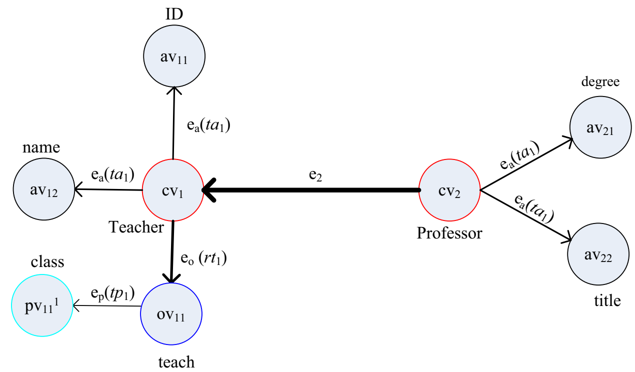


Fig. 10 UCG transformation sample

$$
\begin{aligned}
&\mathit{for}\ UCD = (C, A, O, P, R)\\
&\forall c_i \in C(1 \le i \le n) \Rightarrow \exists cv_i \in CV + L^C(cv_i)\\
&\forall a_{ij} \in A_i(1 \le i \le n) \Rightarrow \exists av_{ij} \in AV_i + L^A(av_{ij})\\
&\quad + ae_{ij}(e_a) \in AE_i\\
&\forall o_{ij} \in O_i(1 \le i \le n) \Rightarrow \exists ov_{ij} \in OV_i + L^O(ov_{ij})\\
&\quad + oe_{ij}(e_o) \in OE_i\\
&\forall p^f_{ij} \in P_{ij}(1 \le i \le n,\ 1 \le j \le |O_i|) \Rightarrow \exists pv^f_{ij} \in PV_{ij}\\
&\quad + L^P\left(pv^f_{ij}\right) + pe^f_{ij}(e_p) \in PE_{ij}\\
&\forall r_{ij}(t_m) \in R(1 \le i,j \le n) \Rightarrow \exists re_{ij}(e_m) \in RE
\end{aligned}
$$

Then,

$$
\begin{aligned}
&AV = \{AV_1, AV_2, \ldots, AV_n\}\\
&OV = \{OV_1, OV_2, \ldots, OV_n\}\\
&PV = \{PV_1, PV_2, \ldots, PV_n\} \text{ and } PV_i = \{PV_{i1}, PV_{i2}, \ldots, PV_{in}\}
\end{aligned}
$$

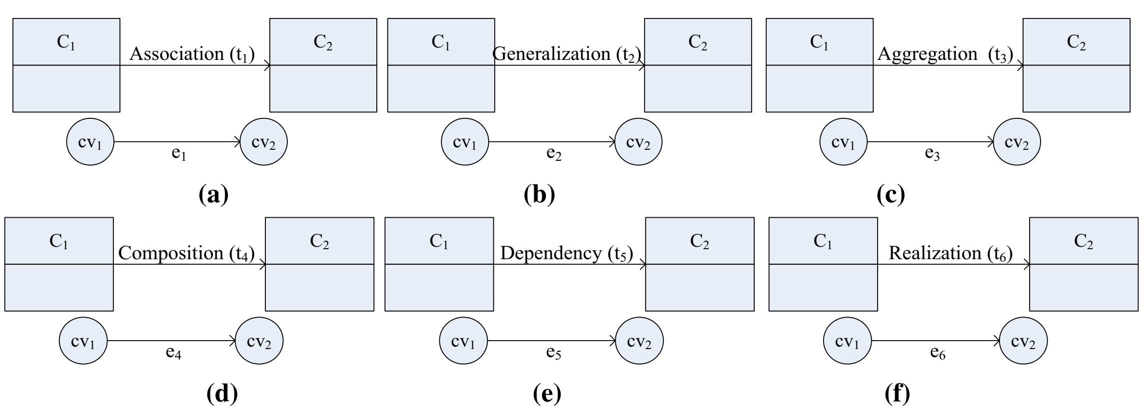


Fig. 9 The direction setting of relationship edges

and

$AE = \{AE_1, AE_2, \dots, AE_n\}$
$OE = \{OE_1, OE_2, \dots, OE_n\}$
$PE = \{PE_1, PE_2, \dots, PE_n\}$ and $PE_i = \{PE_{i1}, PE_{i2}, \dots, PE_{in}\}$

So,

$CV \cup AV \cup OV \cup PV \Rightarrow V$
$AE \cup OE \cup PE \cup RE \Rightarrow E$

and

$L^C + L^A + L^O + L^P \Rightarrow L$

Let,

$(V, E, L) \Rightarrow UCG$

4 Structural similarity measure

The inter-structure of an UCG can be thought of as the structure after deleting attribute vertices (edges), operation vertices (edges) and parameter vertices (edges), corresponding to the mainframe of a class diagram. The inter-structure of an UCG plays a decisive role in the structural similarity measure. The intra-structure of an UCG is expressed by these elements (i.e., attribute vertices, operation vertices and parameter vertices) connecting to a class vertex, corresponding to the composition of a class existing in an UCD.

The structural similarity measure is to quantify the structural difference. The similarity value is limited to [0, 1], where 0 means completely different and 1 means identical. Due to the characteristics that an UCG consists of different types of vertices and edges, the matching and comparing of structure can only be carried out among the elements with the same types. We have some correspondences: class vertex is to class vertex, attribute vertex (edge) is to attribute vertex, operation vertex (edge) is to operation vertex, parameter vertex (edge) is to parameter vertex and relationship edge is to relationship edge. The structural matching is based on the tags of edges, instead of vertices: the same tag indicates the same structure and vice versa. The structural similarity measure between UCG is defined as bellows.

$$Sim(g_1, g_2) = \theta * simInter(g_1, g_2) + (1 - \theta) * simIntra(g_1, g_2) \tag{1}$$

Here *simInter* and *simIntra* denote the similarity of inter-structure and the intra-structure, respectively, and θ is the weighting factor (θ is limited to [0, 1] and usually close to 0.9).

4.1 Preliminary knowledge

Maximum Common Subgraph (denoted as MCS) and Edit Distance (denoted as ED) are frequently used methods for graph isomorphism [36, 37]. UCG maximum common subgraph and UCG edit distance are first proposed in this section and then applied to the inter-structure similarity measure and intra-structure similarity measure, respectively.

4.1.1 UCG maximum common subgraph

Here UCG Maximum Common Subgraph is from the inter-structure of UCG, which is only applied to the inter-structure similarity measure. Obtaining UCG Maximum Common Subgraph is based on the tags of relationship edges, instead of class vertices. Firstly, UCG Maximum Common Subgraph is defined and then UCG Maximum Common Subgraph List and UCG Maximum Common Subgraph Tree are proposed, respectively.

Definition 3 (***UCG Maximum Common Subgraph***) Let ucg_1 and ucg_2 be two UCG. Suppose that there exists an UCG g and there is not an UCG g', where $g \subseteq ucg_1$, $g \subseteq ucg_2$, $g' \subseteq ucg_1$, $g' \subseteq ucg_2$, and $|g'| > |g|$ ($|g|$ is used to denote the number of relationship edges existing in g). Then g is called **UCG Maximum Common Subgraph** (denoted as **UMCS**) between ucg_1 and ucg_2.

Here, the size of an UMCS can be measured by the number of relationship edges existing in UMCS. The number of UMCS may be more than one, especially for UCG with larger size. It is assumed that $g_1, g_2, \dots, g_m$ are UMCS between ucg_1 and ucg_2. Then, these UMCS constitute a list called **UMCS List** (denoted as **UMCSL**) and we have $\mathrm{UMCSL}_1 = \{\mathrm{UMCS}_1^1, \mathrm{UMCS}_2^1, \mathrm{UMCS}_3^1, \dots, \mathrm{UMCS}_m^1\}$, where g_i is denoted as UMCS_i^1. Based on each UMCS_i^1 existing in UMCSL_1, we can obtain UMCSL_2 between (ucg_1–UMCS_i^1) and (ucg_2–UMCS_i^1). That is, $\mathrm{UMCSL}_2 = \{\mathrm{UMCS}_{11}^2, \mathrm{UMCS}_{12}^2, \dots, \mathrm{UMCS}_{m1}^2 \mathrm{UMCS}_{m2}^2 \dots, \mathrm{UMCS}_{mh}^2\}$. This process is repeated until there is not any UMCS between the remainders of ucg_1 and ucg_2. All these UMCSL are inserted into an **UMCS Tree** shown in Fig. 11. UMCS Tree is initialized as a root node and it is empty.

4.1.2 UCG edit distance

The basic idea of graph edit distance comes from string edit distance [38], which is used to find the minimum operation distance while transforming one graph to another. The edit distance between two graphs g_1 and g_2 is defined as follows.

$$\mathrm{GED}(g_1, g_2) = \min_{1 \le j \le m} \sum_{i=1}^{k} {}_{e1,\dots,ek \in p_j(g_1, g_2)} \cos t(ei) \tag{2}$$

Here, cost (e_i) denotes the cost of edit operation e_i and p_j (g_1, g_2) denotes an edit path for transforming g_1 into g_2. There may be multiple edit paths for transforming g_1 to g_2 and the

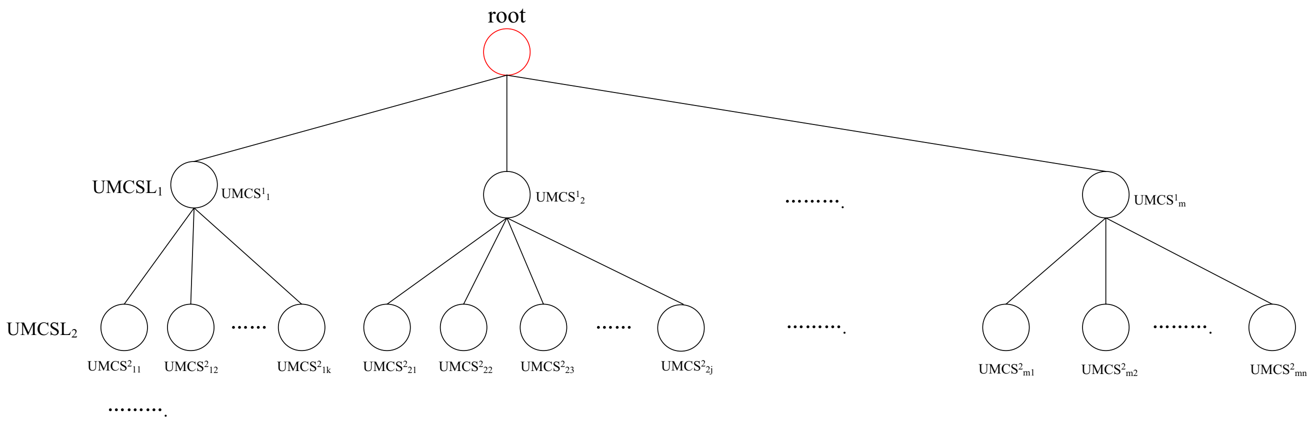


Fig. 11 UMCS tree

Table 2 UCG editing operations

Edit operation		Description	Edit cost
Insertion	Ive_1	Insert an attribute vertex and the corresponding attribute edge	IC_1
	Ive_2	Insert an operation vertex and the corresponding operation edge	IC_2
	Ive_3	Insert a parameter vertex and the corresponding parameter edge	IC_3
Deletion	Dve_1	Delete an attribute vertex and the corresponding attribute edge	DC_1
	Dve_2	Delete an operation vertex and the corresponding operation edge	DC_2
	Dve_3	Delete a parameter vertex and the corresponding parameter edge	DC_3

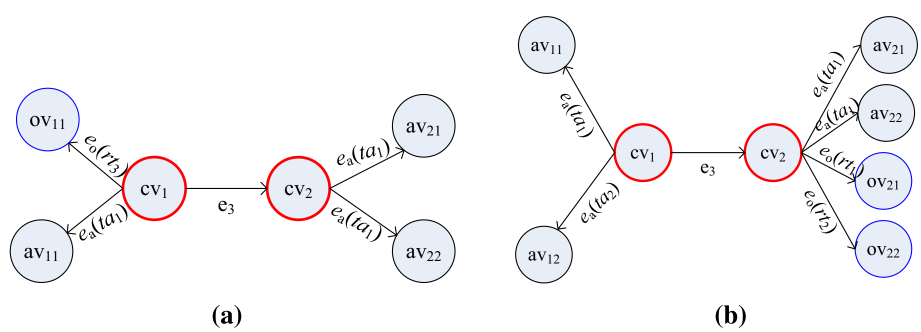


Fig. 12 UCG edit distance case

edit distance is to find the path whose edit cost is the least. A standard set of edit operations generally includes insertion, deletion and substitution of both vertices and edges. In this paper, UCG edit distance is proposed and applied to the intra-structure similarity measure, in which only two operations are allowed: *insertion* and *deletion*. The label of vertex is ignored when the edit distance is calculated. The reason is that we are talking about structure, not semantics. The edit operations of UCG are summarized in Table 2.

On the basis of Table 2, we define the UCG edit distance as follows.

$$\mathrm{UCGED}(g_1, g_2) = x_1 * \mathrm{IC}_1 + x_2 * \mathrm{IC}_2 + x_3 * IC_3 + y_1 * \mathrm{DC}_1 + y_2 * \mathrm{DC}_2 + y_3 * \mathrm{DC}_3 \tag{3}$$

Here, x_1, x_2, x_3, y_1, y_2 and y_3 are some coefficients, which are the times of the corresponding edit operation. Note that the insertion and deletion operations that are applied to the same object are assigned to the same edit cost, that is, $IC_1 = DC_1$, $IC_2 = DC_2$ and $IC_3 = DC_3$. Then the formula above can be further stated as follows.

$$\mathrm{UCGED}(g_1, g_2) = (x_{1+}y_1) * IC_1 + (x_{2+}y_2) * IC_2 + (x_{3+}y_3) * IC_3 \tag{4}$$

Let us look at an example shown in Fig. 12, where the UCG in Fig. 12a is matched to UCG in Fig. 12b. We

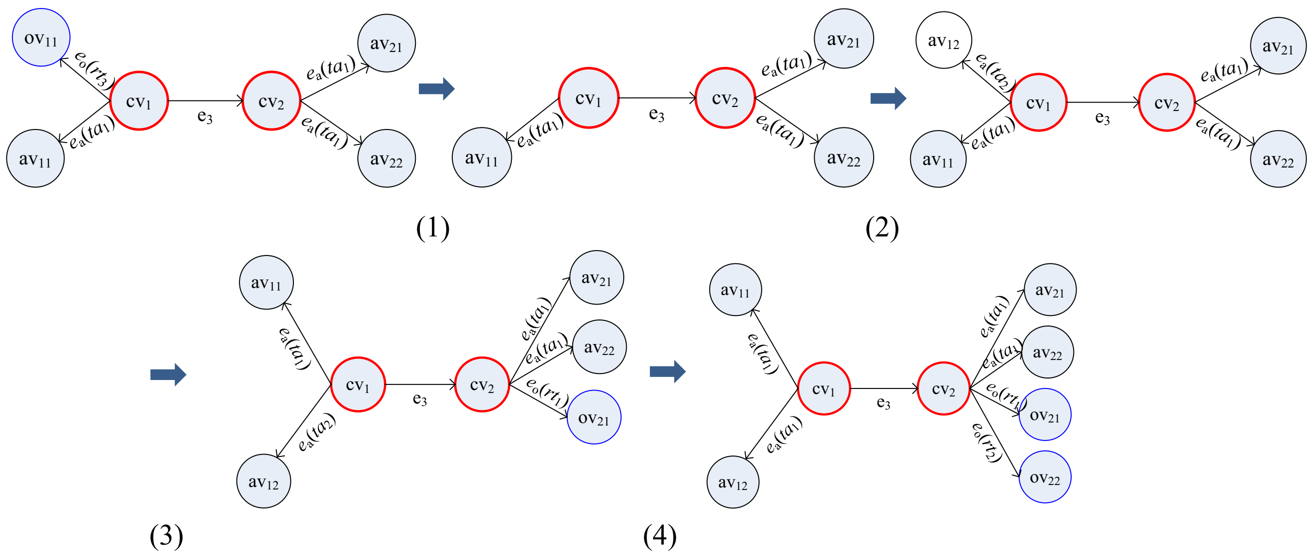


Fig. 13 Editing path from UCG in Fig. 12a to UCG in Fig. 12b

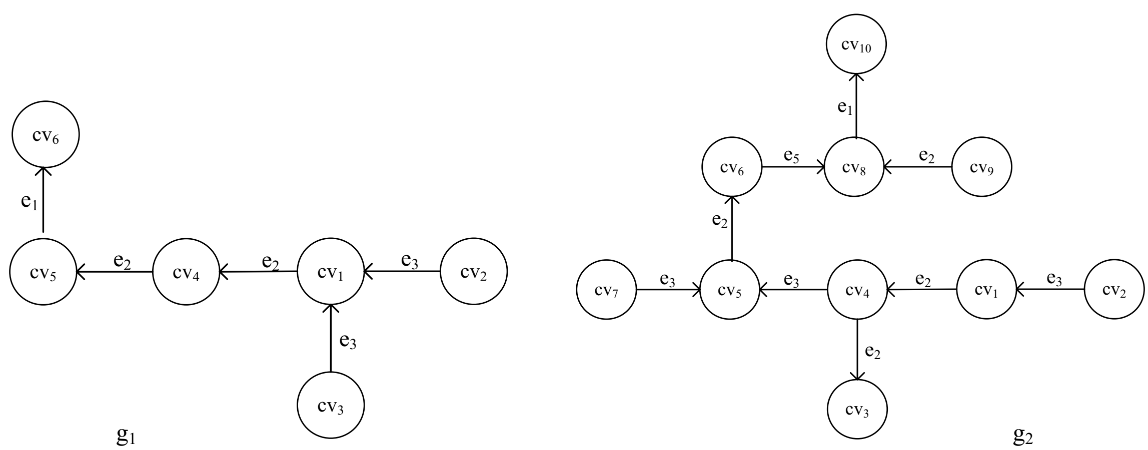


Fig. 14 UCG examples for the structural similarity measure

calculate the edit distance from UCG in Fig. 12a to UCG in Fig. 12b based on the formula (4).

Obviously, after deleting an operation vertex ov_{11} and its corresponding operation edge oe_{11}, inserting an attribute vertex av_{12} and its attribute edge ae_{12} to cv_1, and adding two operation vertices ov_{21} and ov_{22} and their corresponding operation edges oe_{21} and oe_{22} to cv_2, the UCG in Fig. 12a becomes the UCG in Fig. 12b in the structure. The edit path is shown from Step (1) to Step (4) in Fig. 13, where UCG edit distance is UCGED (a, b) = $IC_1 + 3IC_2$.

4.2 Similarity measure

The Similarity is based on the common parts of objects that are matching one another. Let us see an example. Two UCG g_1 and g_2 are transformed from UML class diagrams in an education domain, shown as Fig. 14, they have similar structures. We only show the inter-structure of g_1 and g_2 and the labels of the vertices are removed for saving space. Note that the same tags of class vertices from g_1 and g_2 (e.g., cv_1, cv_2, …, cv_6) do not mean that these vertices are identical. Again, to save space, we do not show the intra-structures and the distributions of attribute vertices (edges) and operation (parameter) vertices (edges) connecting to each class vertex existing in g_1 and g_2 are shown in Tables 3 and 4, respectively. In this section, the inter-structure similarity and the intra-structure similarity are discussed, respectively.

4.2.1 Inter-structure similarity

UMCS Tree provides a solution for using common parts to measure the inter-structure similarity. Each path from the root

Table 3 Distribution of attribute vertices and operation (parameter) vertices in g_1

Number	cv_1	cv_2	cv_3	cv_4	cv_5	cv_6
Attribute vertices	4	4	5	6	4	4
Operation vertices	2	2	2	2	2	2
Parameter Vertices	2	3	4	2	4	2

Table 4 Distribution of attribute vertices and operation (parameter) vertices in g_2

Number	cv_1	cv_2	cv_3	cv_4	cv_5	cv_6	cv_7	cv_8	cv_9	cv_{10}
Attribute vertices	4	6	2	4	5	5	6	6	4	5
Operation vertices	2	3	1	2	2	2	3	2	2	3
Parameter Vertices	2	4	2	3	2	2	5	4	3	4

to a leaf node constitutes an UMCS Sequence (denoted as UMCSS). A preorder traversal of UMCS Tree can obtain all UMCSS. We have $\mathrm{UMCSS}_i=\{\mathrm{UMCS}_j^1, \mathrm{UMCS}_{jp}^2, \ldots, \mathrm{UMCS}_{jp\ldots k}^w\}$, where $|\mathrm{UMCS}_j^1| \geq |\mathrm{UMCS}_{jp}^2| \geq \ldots \geq |\mathrm{UMCS}_{jp\ldots k}^w|$. Then UMCSS_i with the largest number of elements is chosen to measure the inter-structure similarity between two matched UCG, which is defined as follows. Of course, there may be more than one like UMCSS_i.

$$SimInter(ucg_1, ucg_2) = \frac{\max\left(|\mathrm{UMCSS}_1|, |\mathrm{UMCSS}_2|, \ldots, |\mathrm{UMCSS}_n|\right)}{\min\left(|ucg_1|, |ucg_2|\right)} \tag{5}$$

$$|\mathrm{UMCSS}_i| = \sum\nolimits_{\mathrm{UMCS} \in \mathrm{UMCSS}_i} |\mathrm{UMCS}| \tag{6}$$

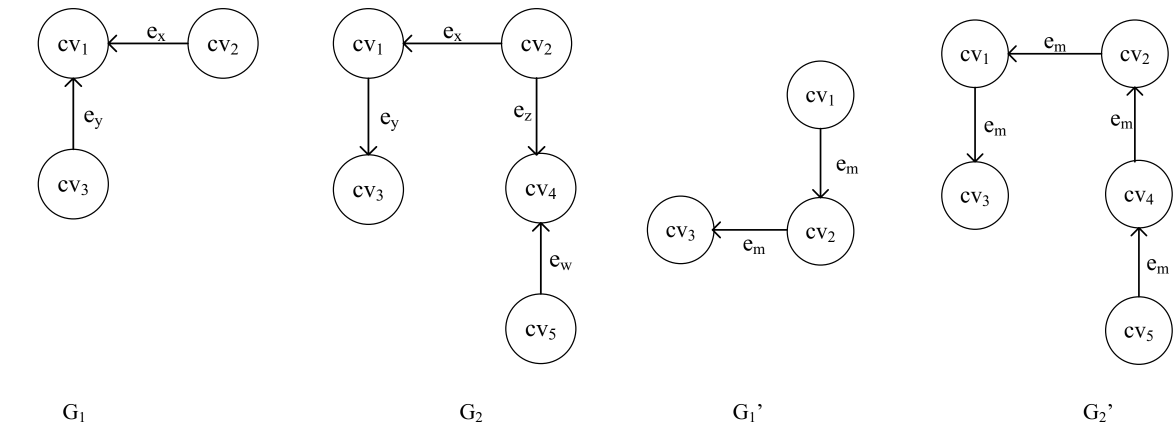
Now, an important task is to create the UMCS Tree. The algorithm of creating UMCS tree is described in Algorithm 1.

```
Algorithm 1. CreateUMCSTree(UMCSNode t, UCG g1, UCG g2)
Input: UCG g1, g2
Output: UMCS tree t
1.  mcsl =getMCSL( t, g1, g2);
2.  if(mcsl!=Null) {
3.    insertUMCSTree(mcsl, t );
4.    for each umcs ∈ mcsl do {
5.     g1=g1- umcs;
6.     g2=g2- umcs;
7.    CreateUMCSTree(umcs, g1, g2 );
8.  }
9.  else
10.   return t;
```

UMCS Tree t is initialized as a root node and it is NULL. The *mcsl* is used to store UMCSS between g_1 and g_2 in Step 1. The construction of UMCS tree is a process of repeatedly obtaining UMCSL and inserting it into UMCS tree from Step 1 to Step 7 until there is not any UMCSL in Step 10. This process is a recursion. It can be seen from Algorithm 1 that, to create UMCS tree, we need to achieve UMCSL first and we propose Algorithm 2 to deal with the issue.

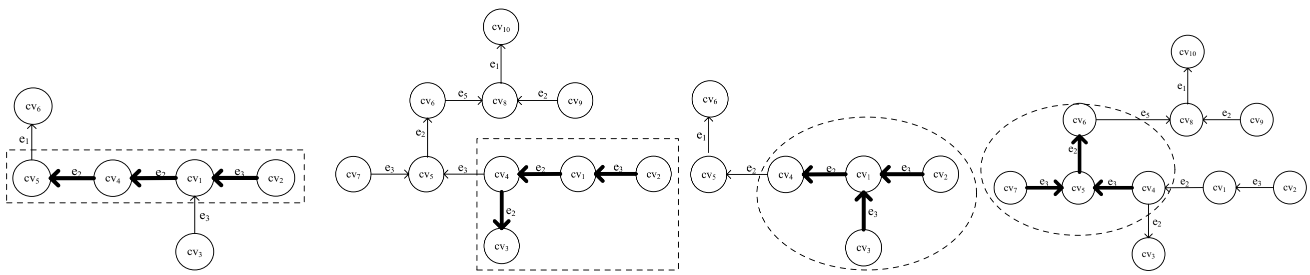
```
Algorithm 2. Search UMCSL between g1 and g2
Input: UCG g1, g2
Output: UMCSL mcsl
1. mcsl=Null;
2. S=Null;
3. while (nextRE(g1, S, reij)) do {
4.   if(IsFeasibleRE(g1, g2, S, reij)) {
5.     S=S+ reij;
6.     if(size(S)>currentSize){
7.      saveCurrentMCS(S);
8.      currentSize=size(S);
9.      clearMCSL(mcsl);
10.       insertMCSL(S, mcsl);
11.   }
12.   else if ((size(S) = currentSize) and (S not in mcsl)) {
13.       appendMCSL(S, mscl);
14.   }
15. }
16. else
17.   backState(S);
18. }
19. return mcsl;
```

Algorithm 2 performs a depth-first searching. Here S is a state space that stores common subgraph between g_1 and g_2 under construction and is a fragment of UMCS to be formed. We may have more than one UMCS and so *mcsl* is used to store all UMCS. S and *mcsl* are initialized as empty (Step 1 and Step 2). Then a relationship edge re_{ij} from g_1 is added to S. It is necessary to check if it is possible to extend the common subgraph represented by an actual state S by the means of adding the relationship edge re_{ij} to S. If this extension is successful, a new state space S replaces the old one. If the current partial solution is larger than the stored solution, it becomes the new stored solution and is inserted into *mcsl* (Step 4 to Step 11). *saveCurrentMCS*, *clearMCSL* and *insertMCSL* are three functions, which save UMCS to *mcsl*, clear *mcsl* and insert UMCS to *mcsl*, respectively. If the size of current partial solution



(a) The best case **(b)** the worst case

Fig. 15 The inter-structure similarity cases



(a) $\mathrm{UMCS}^1{}_1$ **(b)** $\mathrm{UMCS}^1{}_2$

Fig. 16 UMCSL_1

is equal to the stored solution and the current partial solution is not contained in *mcsl*, it is appended to *mcsl* as another UMCS (Step 12 to Step 13) and then next UMCS is continuously searched. *backState*(S) is used to restore the previous state of S in Step 17.

It is well known that obtaining MCS between two graphs is a NP problem, but the actual computation time is still acceptable in many applications. The reason is based on the fact that the graphs encountered in practice are usually different from the worst cases existing in general graphs. For an UCG, the characteristics of nodes and edges can be used very often to reduce the searching time dramatically [39]. Figure 15 gives the best and worst cases that may occur in the inter-structure similarity measure.

In a best case, each relationship edge of G_1 is perfectly matched only to the relationship edge of G_2, which is shown in Fig. 15a, and UMCS is easily obtained. A worst case shown as Fig. 15b is that all relationship edges existing both in G_1 and G_2 have the same tags. At this point, an UCG is evolved into a general digraph and obtaining UMCS becomes a NP problem. It should be noted that it is almost impossible that such a worst case could occur. This is because that UCG is transformed from UCD, and it is impossible that all relationships of UCD are the same. Generally, the average number of class vertices of an UCG is not more than 30 [40]. So, an UCG is not a large graph and the time complexity of the worst case is not too bad. The basic idea of obtaining UMCS in this paper mainly comes from McGregor [36]. The difference of our approach is that our searching UMCS starts from edge instead of vertex.

Now, we begin to calculate the inter-structure similarity between g_1 and g_2 in Fig. 14 based on the proposed algorithm. We need to create an UMCS tree. An UMCS tree is initialized as a root node, and it does not contain any vertices and edges. The specific process is as follows:

(1) Obtaining UMCSL_1 between g_1 and g_2

Two UMCS between g_1 and g_2 can be obtained, which are shown in Fig. 16 as (a) UMCS_1^1 and (b) UMCS_2^1 circled with a dotted rectangle and ellipse, respectively. We have $\mathrm{UMCSL}_1 = \{\mathrm{UMCS}_1^1, \mathrm{UMCS}_2^1\}$. All these elements in UMCSL_1 are inserted into UMCS tree.

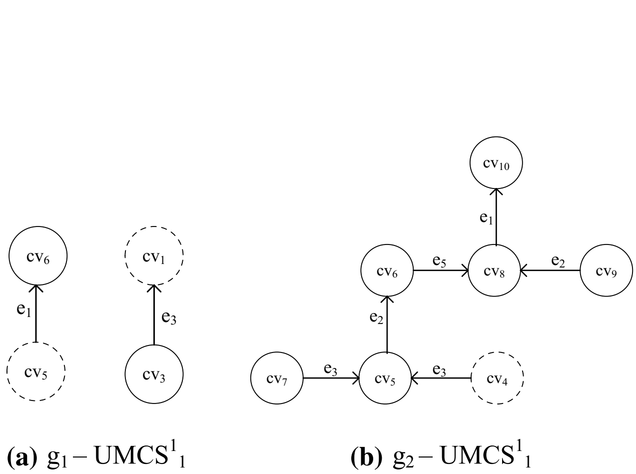


Fig. 17 The remainders of g_1 and g_2

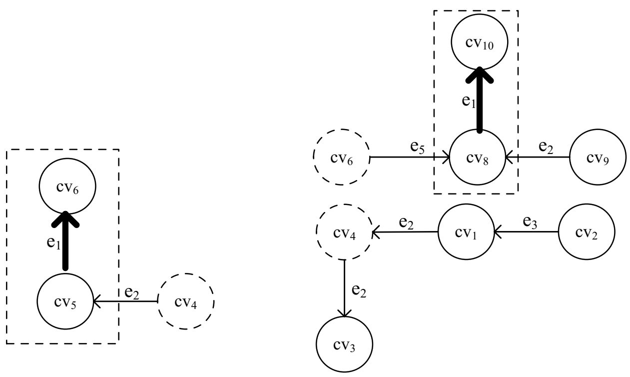


Fig. 18 UMCS_{21}^{2}

(2) Searching UMCSL_2 between the remainders of g_1 and g_2

Then g_1—UMCS_1^1 and g_2—UMCS_1^1 as well as g_1—UMCS_2^1 and g_2—UMCS_2^1 are shown in Fig. 17, respectively.

The vertices marked by dotted lines become the part of the exited UMCS, such as cv_1 and cv_5 in Fig. 17a. The existence of a relationship edge depends on two class vertices at each end. Obviously, there is not a complete relationship edge in g_1—UMCS_1^1, but there are still a few relationship edges to be not matched, which emerge in g_2—UMCS_1^1 and are shown in Fig. 17b. So, UMCS between g_1—UMCS_1^1 and g_2—UMCS_1^1 does not exist. UMCS between g_1—UMCS_2^1 and g_2—UMCS_2^1 can be easily found, it is circled with a dotted rectangle and denoted as UMCS_{21}^2 in Fig. 18. That is, $\mathrm{UMCSL}_2=\{\mathrm{UMCS}_{21}^2\}$. Then, the searching process can finally stop because there is not a relationship edge in the remainders of g_1—UMCS_2^1—UMCS_{21}^2. As shown in Fig. 19, the element in UMCSL_2 is also inserted into UMCS tree.

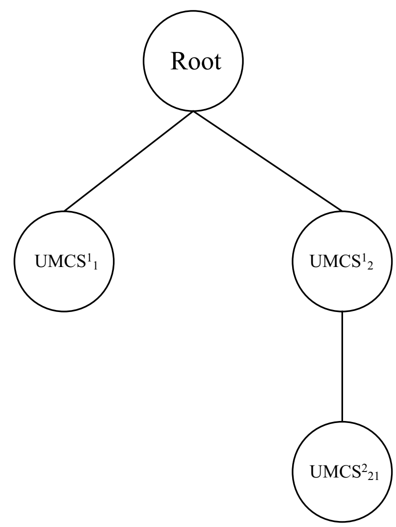


Fig. 19 UMCS tree

Obviously, two paths exist in the UMCS tree: $\mathrm{UMCSS}_1=\{\mathrm{MCS}_1^1\}$ and $\mathrm{UMCSS}_2=\{\mathrm{UMCS}_2^1, \mathrm{UMCS}_{21}^2\}$, where $|\mathrm{UMCSS}_2|>|\mathrm{UMCSS}_1|$. That is, the inter-structure similarity between g_1 and g_2 can be measured by UMCSS_2. We use the formulas (5) and (6) to calculate the inter-structure similarity as follows.

$$SimInter(g_1, g_2) = \frac{\left|\mathrm{UMCS}_2^1\right|+\left|\mathrm{UMCS}_{21}^2\right|}{\min(|g_1|, |g_2|)} = (3+1)/5 = 0.80$$

The corresponding class vertices matching pairs in the inter-structure similarity are described in Table 5.

Here the same tag emerges in the relationship edges re_{21} and re_{31} of g_1. So, the matching pair 2 and 3 can be adjusted from $g_1.cv_2$ to $g_2.cv_7$ and from $g_1.cv_3$ to $g_2.cv_4$.

Table 5 Class vertices matching pairs in the inter-structure similarity

Matching pair	g_1	g_2
1	cv_1	cv_5
2	cv_2	cv_4
3	cv_3	cv_7
4	cv_4	cv_6
5	cv_5	cv_8
6	cv_6	cv_{10}

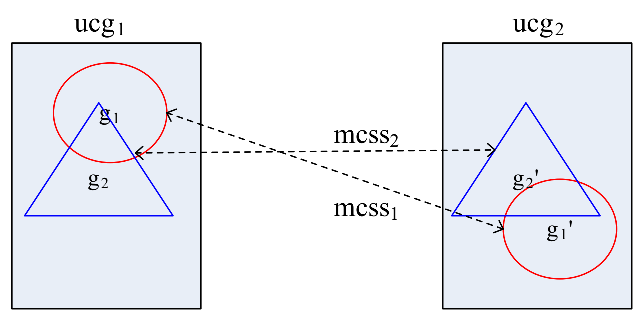


Fig. 20 MCSS cases

4.2.2 Intra-structure similarity

Frequently, there are more than one UMCSS that satisfies the same inter-structure similarity values. For example, there are $umcss_1$ and $umcss_2$ between ucg_1 and ucg_2 and the same values can be obtained by using $umcss_1$ and $umcss_2$ to calculate the inter-structure similarity, shown as Fig. 20, where $|umcss_1| = |umcss_2|$. At this point, choosing which one of $umcss_1$ or $umcss_2$ as the final answer of the inter-structure similarity is decided by the intra-structure similarity.

In this paper, we introduce UCG edit distance discussed in Sect. 4.1.2 to the intra-structure similarity measure. The intra-structure similarity is based on the inter-structure similarity. The intra-structure similarity is captured from three aspects: attribute vertex (edge), operation vertex (edge) and parameter vertex (edge). To limit the intra-structure similarity value to [0, 1], the intra-structure similarity is defined as follows.

$$
\begin{aligned}
SimIntra\left(g_1, g_1'\right) = {} & \alpha * \left(1 - \frac{\left(x_1 + y_1\right) * \mathrm{IC}_1}{\sum_{mcsg_i \in g_1, mcsg_j \in g_1'} \sum_{\mathrm{AV}_i \in mcsg_i, \mathrm{AV}_j \in mcsg_j} \max\left(\left|\mathrm{AV}_i\right|, \left|\mathrm{AV}_j\right|\right)}\right) \\
& + \beta * \left(1 - \frac{\left(x_2 + y_2\right) * \mathrm{IC}_2}{\sum_{mcsg_i \in g_1, mcsg_j \in g_1'} \sum_{\mathrm{OV}_i \in mcsg_i, \mathrm{OV}_j \in mcsg_j} \max\left(\left|\mathrm{OV}_i\right|, \left|\mathrm{OV}_j\right|\right)}\right) \\
& + \gamma * \left(1 - \frac{\left(x_1 + y_1\right) * IC_1}{\sum_{mcsg_i \in g_1, mcsg_j \in g_1'} \sum_{\mathrm{OV}_i \in mcsg_i, \mathrm{OV}_j \in mcsg_j} \sum_{\mathrm{PV}_{ik} \in \mathrm{OV}_i, \mathrm{PV}_{jw} \in \mathrm{OV}_j} \max\left(\left|\mathrm{PV}_{ik}\right|, \left|\mathrm{PV}_{jw}\right|\right)}\right)
\end{aligned}
\tag{7}
$$

Here, g_1 and $g_1^{'}$ are a matching pair in UMCSS_i and they are from ucg_1 and ucg_2, respectively. Parameters α, β and γ are the weighting factor ($\alpha + \beta + \gamma = 1$), identifying the weight of each part in the intra-structure similarity. Generally, α is close to β and they are all above γ. They are determined by the importance of attributes, operations and parameters contained in a class. The edit cost of all these operations is set to 1, $IC_1 = 1$, $IC_2 = 1$ and $IC_3 = 1$. That is, the edit distance is measured only by the times of the specified edit operation.

In the following, we use the formula 7 to calculate the intra-structure similarity of UMCSS_2 of Fig. 19, we have the following results.

$$
\begin{aligned}
simIntra\left(g_1, g_2\right) &= 0.4 * 0.8065 + 0.5 * 0.8571 \\
&\quad + 0.1 * 0.8500 = 0.8362
\end{aligned}
$$

Here, α, β and γ are set to 0.4, 0.5 and 0.1, respectively. When the matching pair 2 and 3 is adjusted according to the above statements, another intra-structure similarity value can be calculated, and it is 0.7895. Obviously, the matching pair that is combined with a larger similarity value 0.8362 is accepted. The final structural similarity value between g_1 and g_2 is:

$$
Sim\left(g_1, g_2\right) = 0.90 * 0.8000 + 0.10 * 0.8362 = 0.8036
$$

Here, the weighting factor θ is set to be 0.9.

5 Experiment

In this section, we design an experiment to evaluate our proposed approach. A prototype system was developed, which was implemented using Java and run on a computer (CPU I5 2.5G, RAM 8G) using Windows 7. We use Microsoft SQL Server 2008 to store UML class diagrams for our experiment. We use the experiment to prove that:

Table 6 The description of class diagrams used in the experiment

UCD	Modeling field		Number	Average size	Category
Query class diagrams	Education		5	12	QC_1
	Education		5	23	QC_2
Target class diagrams	1	Education	15	18	TFC_1
		Company	15	17	TFC_2
	2	Education and Company	15	13	TSC_1
		Education and Company	15	23	TSC_2

(1) our proposed approach is suitable for UML class diagrams with various sizes,
(2) our proposed approach is not limited by the modeling field, and
(3) our proposed approach is more accurate than other methods.

5.1 Experimental Data

The class diagrams used in the experiment are from projects developed by software companies, which are divided into two parts: *query class diagrams* and *target class diagrams*. We calculate the structural similarity values between query class diagrams and target class diagrams. The description of the class diagrams used in the experiment is shown in Table 6.

All query class diagrams are from the same domain "Education," and they are classified into two categories based on the size. The sizes of the query class diagrams existing in the first category denoted as QC_1 vary from 10 to 15, and the size of each query class diagram in the second category denoted as QC_2 is limited to 20–25. The number of query class diagrams in both categories is 5. The target class diagrams are partitioned from two different perspectives. Viewed from the modeling field, the target class diagrams are divided into two categories and the number of the class diagrams is 15 in each category. In the first category denoted as TFC_1, all target class diagrams are from "Education" and describe the same or similar projects as query class diagrams. In the second category denoted as TFC_2, the modeling field of target class diagrams is from "Company," which is completely different from the first category but still similar in structure. Viewed from the size of the target class diagrams, they can be divided into two categories and the number of class diagrams in each category is 15. The size of each target class diagram from the first category denoted as TSC_1 is limited to 10–15, and the sizes of target class diagrams from the second category denoted as TSC_2 vary from 20 to 25.

5.2 Results analysis

In the experiment, we applied three structure (relationship) similarity measure methods, which are *semantics-based*

Fig. 21 Structural similarity between QC_1 and TFC_1

relationship matching (*Semantics* for short), *model query language-based pattern matching* (*Query Language* for short) and our proposed approach (MCSS for short), respectively. The first two methods have been mentioned in [15, 27]. Each query class diagram is matched to all target class diagrams, and all the structural similarities are calculated by these three methods. In our proposed MCSS, the weighting factors θ, α, β and γ are set to 0.9, 0.4, 0.5 and 0.1, respectively. In the *semantics*-based method, the weights of relationship type and end class are set to 0.5 and 0.5 when the relationship is matched.

To assess these three methods, we also invited five experts who are software engineers with rich experience in software design. The experts were requested to compare the query class diagrams and target class diagrams and then answer the same problem for each comparison between a query class diagram and a target class diagram: "*how structurally similar are these two class diagrams?*". Each expert provided a certain value in [0, 1] for a comparison to identify the structural similarity degree of two compared class diagrams. Here 0 means that two compared models are completely different and 1 means the completely identical. Given that there are two categories of query class diagrams with total 10 query models and 30 target models, each expert made 300 comparisons. Finally, we compared the results obtained by the three methods with the results given by the experts. To avoid listing large amounts of data, the similarity values that a set of query class diagrams are matched to a target class diagram are averaged.

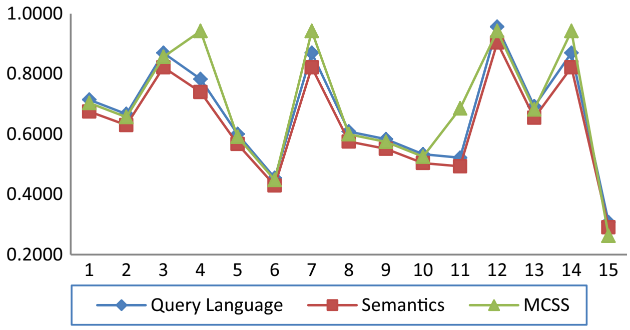


Fig. 22 Structural similarity between QC_2 and TFC_1

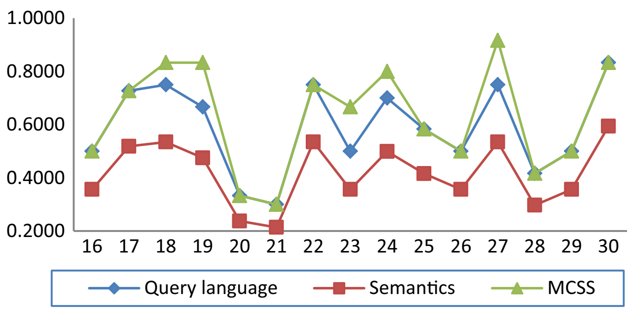


Fig. 23 Structural similarity between QC_1 and TFC_2

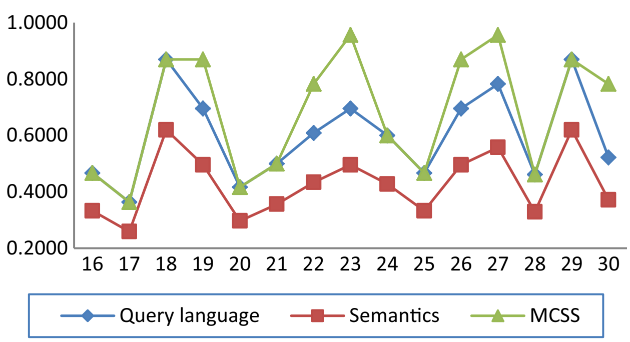


Fig. 24 Structural similarity between QC_2 and TFC_2

For the query class diagrams and the target class diagrams from the same modeling field, shown in Figs. 21 and 22, the results obtained by these methods are close, except for individual values, which is easy to be understood because query class diagrams and target class diagrams describe the same or similar projects, the most structural similarity values are high (≥ 0.5), and only few structural similarity values are low (≤ 0.3). In particular, it is shown in Fig. 21 that the structural similarity values are almost same, which can be explained by the small size of query class diagrams resulting in no common substructures in addition to maximum common substructure in the same modeling field.

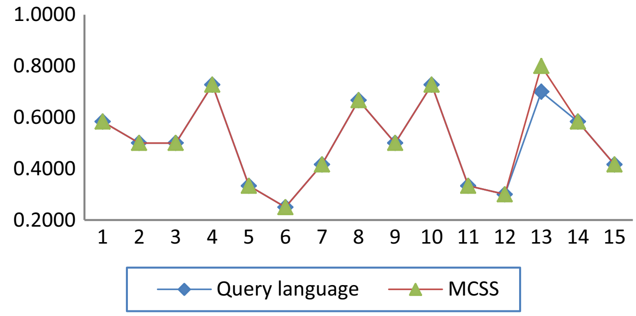


Fig. 25 Structural similarity between QC_1 and TSC_1

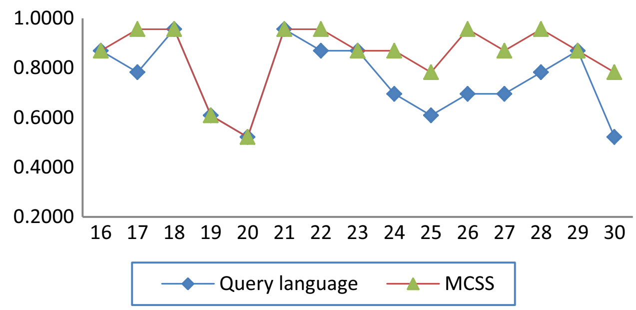


Fig. 26 Structural similarity between QC_2 and TSC_2

It is shown in Figs. 23 and 24 that, however, the results obtained by these three methods have significant differences for different modeling fields. The results obtained by the *semantics* method are significantly smaller than the results obtained by other two methods. The reason is that the *semantics* method considers both relationship type and end class when a relationship is matched, the low semantic similarity between two class names from different modeling domains results in low similarity values and most structural similarity values obtained by the *semantics* method are low (≤ 0.5). Therefore, the *semantics* method is severely affected by the modeling field, but the *semantics* method gives the almost same results as *query language* method when query class diagrams and target class diagrams are from the same domain, regardless of the size of the class diagram being matched.

However, the *query language* method is affected by the size of the class diagrams being matched. When the size of the matched class diagrams is small and close, it is shown in Fig. 25 that the results obtained with *query language* and MCSS method almost has the same results. It is shown in Fig. 26 that, however, the results obtained with these two methods have significant differences for the matched class diagrams in large size, and the values obtained with MCSS are higher than the results obtained with the *query language* method in some matching class diagrams pairs. The reason is that the more common substructures existing between the

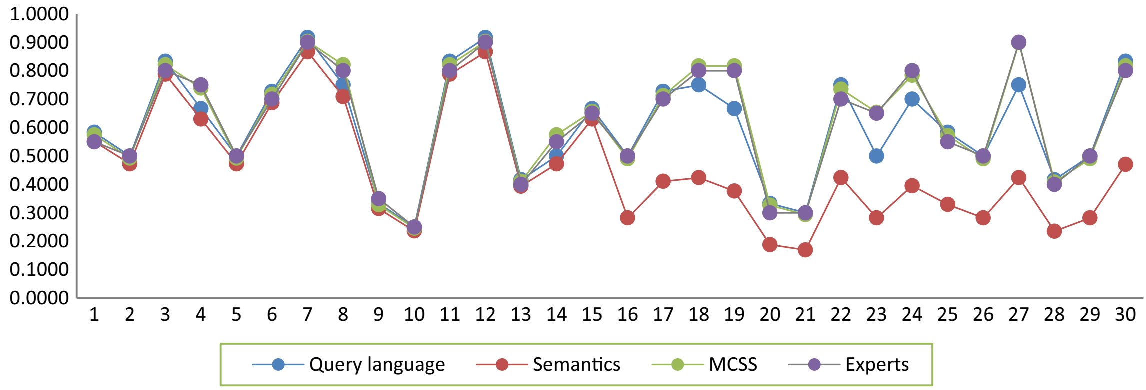


Fig. 27 Structural similarity between QC_1 and $(TFC_1 + TFC_2)$

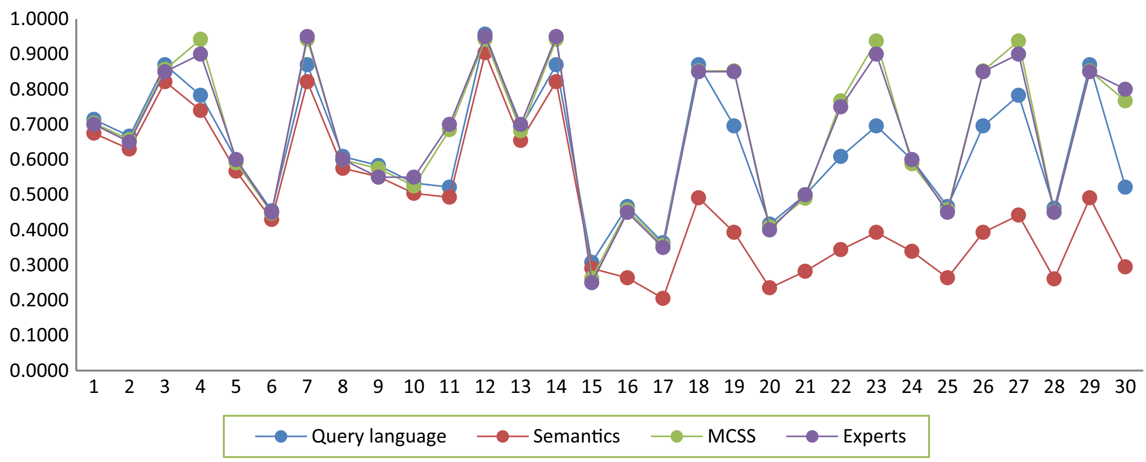


Fig. 28 Structural similarity between QC_2 and $(TFC_1 + TFC_2)$

matched class diagrams are considered in MCSS, in addition to the maximum common substructure which is considered in the *query language* method. Here the results by the semantics-based method are not shown and the reason is that the semantics-based method is affected by the modeling domain rather than the size of class diagrams.

It is shown from the above experimental results that our proposed algorithm is applicable for UML class diagrams with any size and modeling field. As shown in Figs. 27 and 28, no matter which way you look at it, the results obtained by our proposed MCSS are closer to the results given by the experts.

6 Conclusions

In software reuse, the reuse of UML class diagram produced in design phase becomes a major concern. The existing works on the reuse of class diagram mainly focus on its semantic reuse, and its structural reuse is rarely noticed. This paper proposes reusing class diagrams in another light, namely, structure. The core of the structural reuse is the structural similarity measure. In this paper, we propose to use UML class graph to represent UML class diagram for the purpose of structural similarity measure. The structure is considered from two aspects: inter-structure and intra-structure. An algorithm-based UMCSS is proposed for the inter-structure similarity, and the UCG edit distance is proposed and applied to the intra-structure similarity. The experimental results show that our proposed method is effective and closer to the results given by experts. Note that here we do not mean that this can become a paradigm in conceptual modeling, which is only a way available for conceptual modeling.

In our future work, we will investigate several issues. First, how to improve the efficiency of measuring similarity is one important concern. In this direction, filtering some feature values may help us to do less comparison because of the characteristics of UML class diagram consisting of various relationships. Second, trying other methods (e.g., unit structural matching) is a problem we will consider. UML class graph can be split into pieces of unit structures. On the basis of unit structures, we can obtain the final structural similarity through merging unit structure similarity. Third, transforming UML class diagram into other data models (e.g., XML model) may be a possible way for the structural similarity measure. Finally, in order to improve the matching accuracy, we will consider combining the structural similarity and the semantic similarity together for the reuse.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (61772269 and 61370075).

References

1. Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183
2. Prieto-Diaz R (1993) Status report: software reusability. IEEE Softw 10(3):61–66
3. Prieto-Diaz R (1993) Software reuse: issues and experiences. Am Progr 6(8):10–18
4. Mili H, Mili F, Mili A (1995) Reusing software: issues and research directions. IEEE Trans Softw Eng 22(6):528–562
5. Kim Yongbeom, Stohr Edward A (1998) Software reuse: survey and research directions. J Manag Inf Syst 14(4):113–147
6. Medvidovic N et al (2002) Modeling software architectures in the unified modeling language. ACM Trans Softw Eng Methodol 11(1):2–57
7. Arango G, Schoen E, Pettengill R (1993) Design as evolution and reuse. In: Proceedings of the second international workshop on advances in software reuse, pp 9–18
8. Ali FM, Du W (2004) Toward reuse of object-oriented software design models. Inf Softw Technol 46(15):499–517
9. Adamu A, Zainon WMNW (2016) A review of UML model retrieval approaches. Indian J Sci Technol 9(46):384–390
10. Object Management Group, Unified Modeling Language: Superstructure V2.0, 2005
11. Reiss SP (2009) Semantics-based code search. In: Proceedings of the 31st international conference on software engineering, IEEE Computer Society, IEEE, 2009, pp 243–253
12. Kim J et al (2010) Towards an intelligent code search engine. In: Proceedings of the twenty-fourth AAAI conference on artificial intelligence, pp 1358–1363
13. Alnusair A, Zhao T (2010) Component search and reuse: an ontology-based approach. In: Proceedings of 2010 IEEE international conference on information reuse and integration, pp 258–261
14. McMillan C et al (2012) Exemplar: a source code search engine for finding highly relevant applications. IEEE Trans Softw Eng 38(5):1069–1087
15. Robles K et al (2012) Towards an ontology-based retrieval of UML Class Diagrams. Inf Softw Technol 54(1):72–86
16. Salami HO, Ahmed M (2013) Class diagram retrieval using genetic algorithm. In: Proceedings of 12th international conference on machine learning and application, vol 2, pp 96–101
17. Al-Khiaty MAR, Ahmed M (2014) Similarity assessment of UML class diagrams using a greedy algorithm. In: Proceedings of 2014 international computer science and engineering conference (ICSEC2014), IEEE, 2014, pp 228–233
18. Al-Khiaty MAR, Ahmed M (2014) Similarity assessment of UML class diagrams using simulated annealing, In: Proceedings of 2014 5th international conference on software engineering and service science, IEEE, 2014, pp 19–23
19. Al-Khiaty MAR, Ahmed M (2016) UML class diagrams: similarity aspects and matching. Lect Notes Softw Eng 4(1):41–47
20. Oksana N et al (2015) An approach to compare UML class diagrams based on semantical features of their elements. In: Proceedings of the tenth international conference on software engineering advances, pp 147–153
21. Gomes P et al (2004) Using WordNet for case-based retrieval of UML models. AI Commun 17(1):13–23
22. Miller G (1998) WordNet: an electronic lexical database. MIT press, Cambridge
23. Kara S et al (2012) An ontology-based retrieval system using semantic indexing. Inf Syst 37(4):294–305
24. Cordi V, Lombardi P, Martelli M, Mascardi V (2005) An ontology-based similarity between sets of concepts. In: Proceedings of WOA, pp 6–21
25. Meng L, Huang R, Junzhong G (2013) A review of semantic similarity measures in wordnet. Int J Hybrid Inf Technol 6(1):1–12
26. Lucrédio D, Fortes RPM, Whittle J (2012) MOOGLE: a metamodel-based model search engine. Softw Syst Model 11(2):183–208
27. Zhang X, Chen H, Zhang T (2012) An UML model query method based on structure pattern matching. In: Proceedings of international conference on trustworthy computing and services. Springer, Berlin, Heidelberg, vol 320, pp 506–513
28. Qiu DH, Li H, Sun JL (2013) Measuring software similarity based on structure and property of class diagram. In: Proceedings of 2013 sixth international conference on advanced computational intelligence, IEEE, pp 75–80
29. Salami HO, Ahmed M (2014) Retrieving sequence diagrams using genetic algorithm. In: Proceedings of 2014 11th international joint conference on computer science and software engineering, IEEE, pp 324–330
30. Ahmed M, Salami HO (2015) Behavior-based retrieval of software. Afr J Comput ICT 8(1):95–102
31. Routledge N, Bird L, Goodchild A (2002) UML and XML schema. In: Proceedings of 2002 thirteenth Australasian database conference DBLP on database technologies, pp 157–166
32. Grose TJ, Doney GC, Brodsky SA (2002) Mastering XMI Java Programming with XMI, XML and UML, vol 20. Wiley, Hoboken
33. Bondy JA, Murty USR (1976) Graph theory with applications, vol 290. Macmillan, London
34. Bunke Horst (2000) Graph matching: theoretical foundations, algorithms, and applications. Proc. Vision Interface 2000:82–88
35. Conte D et al (2004) Thirty years of graph matching in pattern recognition. Int J Pattern Recognit Artif Intell 18(3):265–298
36. Derek G, Gotlieb CC (1970) An efficient algorithm for graph isomorphism. J ACM 17(1):51–64
37. McKay BD (1981) Practical graph isomorphism. J Symb Comput 60(1):94–112
38. Gao X et al (2010) A survey of graph edit distance. Pattern Anal Appl 13(1):113–129
39. Bunke Horst, Shearer Kim (1998) A graph distance metric based on the maximal common subgraph. Pattern Recognit Lett 19(3–4):255–259
40. Bunke H, Messmer BT (1995) Efficient attributed graph matching and its application to image analysis. In: Proceedings of international conference on image analysis and processing, Springer-Verlag, vol 974, pp 45–55