Verifying Software Change Requirements through Conceptual Class Diagrams Comparison

Pattamaporn Saisim and Twittie Senivongse

Abstract—**An existing software system sometimes needs to be redesigned to accommodate various change requirements. A system analyst gathers new user requirements to analyze software requirements and create a conceptual model of the new version of the system. While certain requirements of the existing software system should remain in the new version of the system, some of them may be dropped and some new requirements are to be added. Since incomplete software requirements will lead to incorrect design of the new system, the system analyst needs to verify that the gathered requirements for the new system are complete, i.e. those that should be retained in the new system are not missing and those that are changed or newly introduced are included. This paper presents a method to help the system analyst to verify change requirements for the new version of the software system. As an initial model created from the new software requirements, the conceptual UML class diagram of the new system is compared with that of the existing system. The comparison algorithm called S-UMLDiff considers similarity of the diagram structure and semantic similarity of names in the two diagrams. The reported similarities and differences between the diagrams can assist the system analyst in reviewing the conceptual model of the new system to verify early on whether it is built upon a complete set of change requirements. The paper also presents an evaluation which shows that the S-UMLDiff algorithm performs well, having precision of 0.88 and recall of 0.94.**

Index Terms—**software change requirement, conceptual class diagram, WordNet**

I. Introduction

SOFTWARE systems need to undergo changes constantly. Changes may either be applied directly to existing systems to add or fix certain functions, or they require the systems to be redesigned and reconstructed. The motivation behind redesigning an existing system can be that there are changes in concepts, processes, or functions within the business domain which necessitate changes in the software system structure.

To redesign the software system, the system analyst restarts the whole development process by eliciting new user requirements to gather change requirements as well as studying the requirement specification of the existing system. While certain requirements of the existing software system should remain in the new version of the system, some of them may be dropped and some new requirements are to be added. The problem that may arise is that the users and the development team may be a different group from those who gave the original requirements and developed the existing system. This may result in the software requirements of the new system being incomplete as the users may forget or even not know of certain functions or data that should be retained, and the new development team may not fully understand the business domain. Since incomplete software requirements will lead to incorrect design of the new system, the system analyst needs to verify that the gathered requirements for the new system are complete.

To help ensure that the new version of the system will be developed according to the correct change requirements, this paper presents an approach to verifying change requirements for a new version of a software system through a comparison of conceptual UML class diagrams. As a conceptual model, a conceptual UML class diagram captures important concepts and relationships as classes and their associations within a business domain [1]. We assume that, since the newly-designed conceptual UML class diagram captures initially the software requirements of the new system, comparing it with the conceptual class diagram of the existing system should help identify the similarities and differences between the two system versions. The system analyst can then review certain aspects of the software requirements 1) whether they are still needed but missing from the new model, 2) whether they are not needed but are still included in the new model, 3) whether they should be added but are missing from the new model, and 4) whether they really are changes that should be made in the new model. In other words, the system analyst can verify that certain requirements that should be retained in the new system are not missing and those that are changed or newly introduced are included. The comparison is done by an algorithm called S-UMLDiff, which is an extension to the UMLDiff algorithm proposed by Xing [2]. S-UMLDiff compares two versions of the conceptual UML class diagrams to analyze name and structural changes between subsequent versions. Unlike UMLDiff, the algorithm is enhanced with the capability to analyze semantic similarity between names in the two versions of the diagram, using

Manuscript received June 7, 2014; revised July 1, 2014.

P. Saisim was with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. She is now with KTB Computer Services, 22/1 Sawai Brown 2 Building, Sukhumvit Soi 1, Klongtoey Nua, Wattana, Bangkok 10110 (email: onzony@gmail.com).

T. Senivongse is with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330 Thailand (corresponding author phone: +66 2 2186996; fax: +66 2 2186955; e-mail: twittie.s@chula.ac.th).

WordNet [3]. Considering name similarity improves the ability of the algorithm to recognize additions, removals, matches, moves, renamings of software model elements from one version to the next.

Section II of this paper discusses background and related work. Section III describes the S-UMLDiff algorithm, with an evaluation of its performance given in Section IV. Section V concludes the paper.

II. Background and Related Work

UML class diagrams [1] are useful in many stages of software system design. In the analysis stage, a class diagram can help the system analyst to understand the requirements of the problem domain and to identify important elements, data, functions, and relationships between elements. The class diagram in this stage is conceptual, having no detailed design for the implementation of the software. A conceptual class diagram contains 1) classes that represent concepts in the business domain, 2) attributes of a class, 3) methods of a class, 4) associations, aggregations, compositions, and generalizations which represent different relationships between classes, and 5) packages that represent groups of related classes and their relationships. Here other details such as data types of attributes, method parameters, and visibility of attributes and methods are not of concern.

Many algorithms to compare UML class diagrams have been proposed for different purposes. There is a possibility to apply one of them to our problem, but the chosen algorithm has to be applicable to the conceptual class diagrams which leave out a number of design details and, at the same time, it should be able to accommodate different kinds of changes that could occur in real-world software. Among the algorithms that we consider is the one by Girschick [4] which detects differences between several modifications of a design class diagram for tracking changes during the development process. Matching of design elements are based on generic graph matching techniques, and the elements that are compared are packages, classes, generalizations, attributes, associations, and operations and their parameters. Detected changes are add, delete, rename, move, clone, and modify property (e.g., visibility, data type, multiplicity, stereotype). A color-coding scheme is used to present different kinds of changes in different colors. The algorithm by Auxepaules et al. [5] is also a graph matching method but is used in an object-oriented modeling learning environment. The algorithm compares a student's diagram with an expert's diagram to give the student relevant feedbacks in modeling exercises. It uses the graph matching algorithm of Sorlin et al. [6] and a string matching algorithm of Giunchiglia et al. [7] which also uses WordNet to determine similarity of names. Detected differences are insert, omit, transfer, replace, modify property, merge, split, and cluster.

We select the UMLDiff algorithm of Xing [2] as a basis for our work since the algorithm considers all model elements in a conceptual class diagram and the kinds of differences that are detectable, i.e., add, remove, rename, move, and modify property of elements, are sufficient for class diagrams at a conceptual level. UMLDiff relies on lexical similarity and structure similarity for recognizing the conceptually same model elements in the two compared versions of the class diagram. It does not take semantics of names into account, and thus it cannot recognize when a model element changes to a different but semantically similar name. In addition, it cannot recognize when a model element changes its type, e.g., an attribute is changed to a class.

III. Conceptual Class Diagrams Comparison

We present the S-UMLDiff (or Semantic-UMLDiff) algorithm to compare the conceptual class diagram of a new system with that of the existing system. The S-UMLDiff algorithm shares with UMLDiff in that it considers lexical similarity of names and structural similarity of model elements. Lexical similarity refers to string similarity of names while structural similarity refers to similarity of containment (i.e., a parent element contains another element as its child) and other relationships (i.e., an element has an association, aggregation, composition, and generalization relationship with another element). S-UMLDiff also enhances UMLDiff by considering semantic similarity of names and change of model element types.

To explain the S-UMLDiff algorithm in details, we use a real-world case of a bank in Thailand as an example. Two versions of a conceptual class diagram are in Figs. 1(a) and 1(b). The differences are circled; for example, the class name *Employee* is changed to *Officer*, the attribute *expiryDate* is added to the class *LoanInfo*, the class *Collateral* is removed, the class *Committee* is added to extend from the class *Officer*, and the attribute *address* in the class *Customer* has its element type changed and becomes the class *Address*.

A. Overview of Difference Analysis

The S-UMLDiff algorithm is supported by an analysis tool developed in Java. The analysis consists of:

1) Transform the two conceptual class diagrams into the XML Metadata Interchange (XMI) format. We use the ArgoUML modeling tool [8] to draw the conceptual models and obtain their representation in the XMI format.

2) Extract model elements. Model elements in the two diagrams (i.e., packages, classes, attributes, and methods) are extracted, together with their names and the relationships that they have with other elements and that the other elements have with them. The relationships include containment, association, aggregation, composition, and generalization.

3) Build a directed graph $G(V, E)$ for each version of the class diagram, where the vertex set V contains the extracted model elements and the edge set E contains the relationships among them. An example of the vertices from the new diagram and relevant edges is shown in Table I.

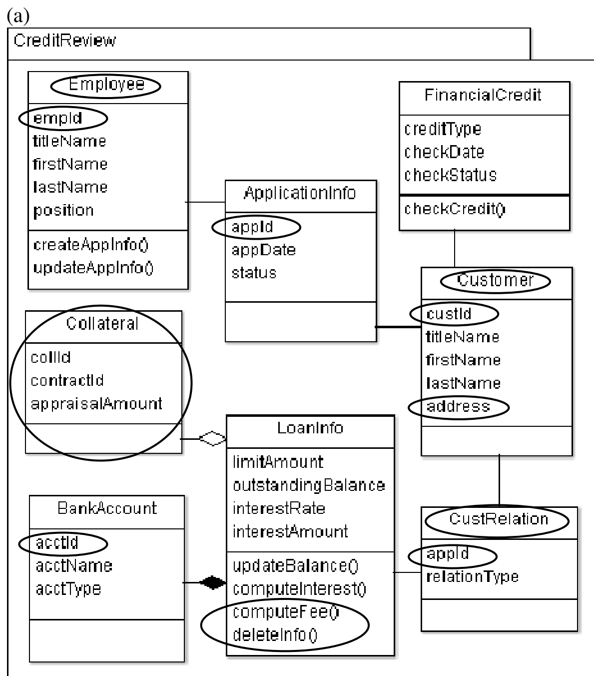
4) Map the two graphs $G_{existing}(V_{existing}, E_{existing})$ and $G_{new}(V_{new}, E_{new})$ by computing the intersection and margin sets between ($V_{existing}$, V_{new}) and ($E_{existing}$, E_{new}) to determine name similarity and structural similarity. That is, ($V_{existing} - V_{new}$) is computed for the removed model elements, ($V_{existing} \cap V_{new}$) for the mapped (i.e., matched, renamed, semantics-

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

of-names-matched, moved, and element-type-changed) elements, ($V_{new} - V_{existing}$) for the added model elements, ($E_{existing} - E_{new}$) for the removed relationships, ($E_{existing} \cap E_{new}$) for the matched relationships, and ($E_{new} - E_{existing}$) for the added relationships. To be precise, the intersection and margin sets are computed by comparing the following in the two diagrams:

4.1) Compare packages;
4.2) Compare classes within the matched packages;
4.3) Compare attributes within the matched classes;
4.4) Compare methods within the matched classes;
4.5) Compare removed class with added attribute, and;
4.6) Compare removed attribute with added class.

(a)



(b)

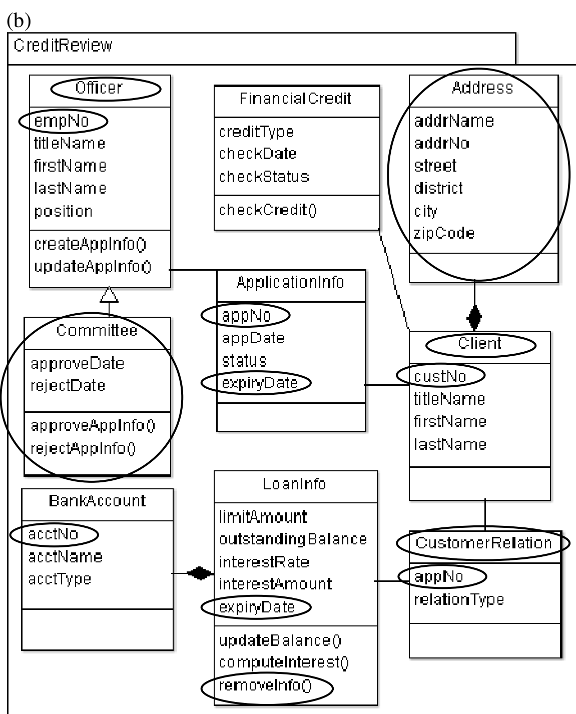


Fig. 1. Example of conceptual class diagrams of (a) existing system (2) new system.

TABLE I
EXAMPLE OF MODEL ELEMENTS AND RELATIONSHIPS FROM NEW DIAGRAM

Relationship	Source		Target	
	Name	Element Type	Name	Element Type
Contain	VirtualRoot[a]	Diagram	CreditReview	Package
	CreditReview	Package	LoanInfo	Class
	LoanInfo	Class	expiryDate	Attribute
	LoanInfo	Class	removeInfo	Method
Generalization	Officer	Class	Committee	Class
Composition	LoanInfo	Class	BankAccount	Class
Association	CustomerRelation	Class	LoanInfo	Class

[a]VirtualRoot is a default name for a conceptual class diagram.

For steps 4.1-4.4, S-UMLDiff identifies:

a) Whether the model elements match by having the same name (i.e., identify Match);

b) Whether the model elements with different names are the case of name change (i.e., identify Rename or Semantic Match) by determining the overall similarity including lexical, semantic, and structural similarities, and;

c) Whether the model elements that are not identified as having a name change are the case of move (i.e., identify Move) by checking if there is a parent change. Otherwise it is the case of add or remove.

Steps 4.5-4.6 are added to S-UMLDiff to determine change of model element types. Types of differences that will be reported by S-UMLDiff are shown in Table II. Details of the comparison are discussed in the subsequent sections.

TABLE II
TYPES OF DIFFERENCES DETECTED IN NEW DIAGRAM

Difference Type	Description
Match	Model element in new diagram is the same as model element in existing diagram.
Rename	The name of model element in new diagram has lexical similarity to a name of model element in existing diagram.
Semantic Match	The name of model element in new diagram has semantic similarity to a name of model element in existing diagram.
Move	The name of model element in new diagram is the same as that of model element in existing diagram but has different parent.
Add	Model element is found in new diagram but not in existing diagram.
Remove	Model element is found in existing diagram but not in new diagram.
Change Type	The name of model element in new diagram is the same as that of model element in existing diagram but has different element type.

B. Name Similarity

To compute similarity of names, S-UMLDiff takes into account lexical similarity and semantic similarity. The model elements in the new conceptual class diagram may use different names for better modeling or due to change of concepts in the problem domain. S-UMLDiff first determines the semantic similarity *wScore* between the two words being compared, using the Wu-Palmer similarity

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

measure that is implemented in the WordNet::Similarity package [3] where *wScore* is in [0, 1]. If the *wScore* is not less than a *Word Similarity Threshold* which is specified by the system analyst, the two words are considered semantically similar. In the case that a string name is not a single word but a phrase (having dots, dashes, underscores and case switching as delimiters between words) and WordNet cannot determine similarity directly, we use a semantic similarity measure for phrases [9]. For phrases *a* and *b* comprising *m* and *n* words respectively, the phrase semantic similarity *pScore* is computed by

$$pScore(a,b) = \frac{\sum_{s=1}^{m} wpScore(a_s,b)}{m} \quad (1)$$

$$wpScore(a_s,b) = \max(wScore(a_s,b_1),...,wScore(a_s,b_n)) \quad (2)$$

where $wScore(a_s, b_n)$ = semantic similarity score between word *s* of phrase *a* and word *n* of phrase *b* by Wu-Palmer measure.

Similarly, if the *pScore* is not less than a *Phrase Similarity Threshold* which is specified by the system analyst, the two phrases are considered semantically similar.

In the case that *wScore* (or *pScore*) is not greater than the corresponding threshold, name similarity is determined by lexical similarity using the Longest Common Subsequence (LCS) algorithm [10]. LCS is the longest subsequence (i.e., a set of characters that appear in left-to-right order but not necessarily consecutively) that appears in both string names *a* and *b*. The lexical similarity metric *lcsScore* is defined by

$$lcsScore(a,b) = \frac{2 * length(LCS(a,b))}{length(a) + length(b)}. \quad (3)$$

For example, using (1) and (2), we can compute the similarity score between the method names *deleteInfo* in Fig. 1(a) and *removeInfo* in Fig. 1(b) by

pScore(*deleteInfo*, *removeInfo*)
= (*wpScore*(*delete*, *removeInfo*) + *wpScore*(*Info*, *removeInfo*))/2
= (*max*(*wScore*(*delete*, *remove*), *wScore*(*delete*, *Info*)) + *max*(*wScore*(*Info*, *remove*), *wScore*(*Info*, *Info*)))/2
= (*max*(0.8, 0) + *max*(0.4, 1))/2 = (0.8 + 1)/2 = 0.9.

Suppose the *Phrase Similarity Threshold* is 0.9, the two phrases are considered similar semantically. But if the threshold is set to 9.5, the two are not similar by semantics and S-UMLDiff will calculate their *lcsScore*.

Note that the part of speech of the two words has to be specified for WordNet::Similarity to obtain *wScore* for them. Since names in the diagrams usually are nouns and verbs and if the two words can be both nouns and verbs, *wScore* for them will be an average of the similarity scores when they are nouns and when they are verbs. In addition, it is assumed that a method name starts with a verb and hence only verb will be used as the part of speech of the first word of a method name.

C. *Structural Similarity*

S-UMLDiff follows UMLDiff in checking for structural similarity by mapping the model elements of the same type and of the same name or similar names and then comparing names of the contained model elements. To compare structure of two packages, their classes are compared. To compare two classes, their attributes, methods, and relationships with other classes are compared. However, in the new version of the diagram, there might be change of model element type such as the attribute *address* of the class *Customer* in Fig. 1(a) is changed to the class *Address* in Fig. 1(b). S-UMLDiff therefore also checks names of class and attribute to see if it is the case of an attribute changing to a class or a class changing to an attribute, and not the case of removed and added model elements.

D. *Overall Similarity*

Computing overall similarity takes into account both names and structure of model elements in the two diagrams.

Let MatchPoint = overall similarity score between model elements *x* and *y*
NamePoint = name similarity score between model elements *x* and *y* (see III.B)
x = model element in the existing diagram
y = model element in the new diagram.

Adapted from UMLDiff, the overall similarity computation for each model element type is as follows.

Overall Similarity between Packages

The following *MatchPoint* between two packages is calculated to determine if they match:

$$MatchPoint = \frac{NamePoint + ChildrenMatchCount}{NamePoint + (xChildrenCount + yChildrenCount - ChildrenMatchCount)} \quad (4)$$

where *ChildrenMatchCount* = number of classes in package *x* whose names match those of classes in package *y*
xChildrenCount = number of classes in package *x*
yChildrenCount = number of classes in package *y*.

Overall Similarity between Classes

The following *MatchPoint* between two classes is calculated to determine if they match:

$$MatchPoint = \frac{NamePoint + ChildrenPoint + UsagePoint}{NamePoint + 3} \quad (5)$$

$$ChildrenPoint = \frac{ChildrenMatchCount}{xChildrenCount + yChildrenCount - ChildrenMatchCount} \quad (6)$$

where *ChildrenMatchCount* = number of attributes in class *x* whose names match those of attributes in class *y* + number of methods in class *x* whose names match those of methods in class *y*
xChildrenCount = number of attributes and methods in class *x*
yChildrenCount = number of attributes and methods in class *y*
UsagePoint = similarity score between names of classes that have relationships with class *x* and names of classes that have relationships with class *y* (see III.B).

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Overall Similarity between Attributes

The following *MatchPoint* between two attributes is calculated to determine if they match:

$$MatchPoint = \frac{ParentPoint * NamePoint}{ParentPoint * NamePoint + 2} \quad (7)$$

where *ParentPoint* = similarity score between class name of attribute *x* and class name of attribute *y* (see III.B).

Overall Similarity between Methods

The following *MatchPoint* between two methods is calculated to determine if they match:

$$MatchPoint = \frac{ParentPoint * NamePoint}{ParentPoint * NamePoint + 2} \quad (8)$$

where *ParentPoint* = similarity score between class name of method *x* and class name of method *y* (see III.B).

Use of MatchPoint

To identify if the two model elements with different names match, i.e., there is a lexical or semantic change of name, the comparison, adapted from UMLDiff, is performed as in Fig. 2. Steps added by S-UMLDiff are shaded.

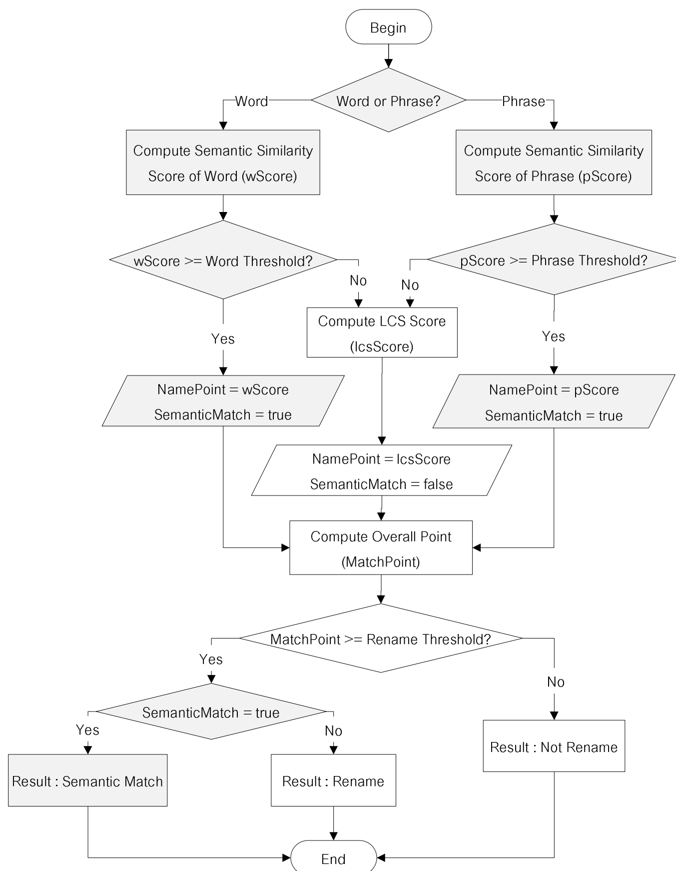


Fig. 2. Comparison to check if two model elements with different names match.

First, semantic similarity of names of the two model elements is determined using *wScore* or *pScore*. As mentioned earlier, if the names are not considered as similar by semantics according to the respective *Word Similarity Threshold* or *Phrase Similarity Threshold*, lexical similarity is computed by *lcsScore*. After that, the overall similarity or *MatchPoint* is computed for the two model elements. In the same manner, the *Rename Threshold* is introduced to identify that the two model elements are a match as they are similar enough by name and structure even though there might be a change of name (i.e. *Semantic Match* or *Rename*). Otherwise, it is the case of *Not Rename* and the two are unmatched, i.e., they are different model elements. For the unmatched elements, they will be checked further for the case of move, add, remove, or change of element type.

Example

This section discusses a comparison between the conceptual class diagrams in Figs. 1(a) and 1(b). Let

match = set of model elements that are considered the same in both diagrams;
first = set of model elements in the existing diagram that remain unmatched, and;
second = set of model elements in the new diagram that remain unmatched.

1) Following the steps in Section III.A.4), first the two packages has the same name *CreditReview* and are considered the same package.

2) Then the classes in the two packages are compared, i.e.,

first = {*FinancialCredit, ApplicationInfo, CustRelation, Employee, Customer, LoanInfo, BankAccount, Collateral*}
second = {*FinancialCredit, ApplicationInfo, Committee, CustomerRelation, Officer, Client, LoanInfo, BankAccount, Address*}.

The comparison of class names results in the following matched and unmatched elements:

match = {(*FinancialCredit:FinancialCredit*), (*ApplicationInfo:ApplicationInfo*), (*LoanInfo:LoanInfo*), (*BankAccount:BankAccount*)}
first = {*CustRelation*, *Employee*, *Customer*, *Collateral*}
second = {*Committee*, *CustomerRelation*, *Officer*, *Client*, *Address*}.

3) Then the classes in *first* and *second* are checked for overall similarity to see if any pairs of classes can match (see III.D). The classes *CustRelation* and *CustomerRelation* are identified as *Rename* and the classes *Employee* and *Officer* as well as the classes *Customer* and *Client* are identified as *Semantic Match*. Therefore the set *match* is updated with these similar classes whereas the classes *Collateral*, *Committee*, and *Address* are identified as *Not Rename* and remain unmatched.

match = {(*FinancialCredit:FinancialCredit*), (*ApplicationInfo:ApplicationInfo*), (*LoanInfo:LoanInfo*), (*BankAccount:BankAccount*), (*CustRelation*:*CustomerRelation*), (*Employee*:*Officer*), (*Customer*:*Client*)}
first = {*Collateral*}
second = {*Committee*, *Address*}

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

4) Next, the classes in *first* and *second* are checked for the case of move, i.e., whether there is a class *Collateral* in other package (i.e., having other parent) in the existing diagram and whether *Committee* and *Address* appear in other package in the new diagram. In this case, there is no other package and so it is not the case of move. Therefore *Collateral* is removed, and *Committee* and *Address* are added.

5) After a comparison at the class level, S-UMLDiff will compare attributes of each pair of matched classes in the set *match*. The comparison is similar to comparing classes, i.e., checking for a match, checking for a name change (semantic match or rename), and checking for move. Likewise, S-UMLDiff will compare methods of each pair of matched classes in the set *match*. The results are

removed class = {*Collateral*}
removed attribute = {*collId, contractId, appraisalAmount, appId, appId, empId, address*}
added class = {*Committee*, *Address*}
added attribute = {*approveDate, rejectDate, addrName, addrNo, street, district, city, zipCode, appNo, appNo, expiryDate, expiryDate, empNo*}
added method = {approveAppInfo, rejectAppInfo}.

6) In addition, S-UMLDiff checks for the case of change of element type and finds that there is a match between the removed attribute *address* and the added class *Address*. Therefore the attribute *address* is of the case of change type.

IV. Evaluation

Evaluation of the S-UMLDiff algorithm is by measuring precision and recall [11] of diagram differences that are reported by the algorithm against the differences identified by a system analyst with 12 years of experiences:

$$precision = \frac{|M_{actual} \cap M_{reported}|}{|M_{reported}|} \tag{9}$$

$$recall = \frac{|M_{actual} \cap M_{reported}|}{|M_{actual}|} \tag{10}$$

where M_{actual} = set of differences identified by the system analyst
$M_{reported}$ = set of differences reported by the algorithm.

Ten pairs of the two versions of the conceptual class diagrams which cover all kinds of changes are used in the evaluation. We adjust the thresholds in the experiment so that they give the best measurement results as shown in Table III. The *Word Similarity Threshold*, *Phrase Similarity Threshold*, and *Rename Threshold* are 0.7, 0.9, and 0.5 respectively. The algorithm gives the average precision of 0.88 and average recall of 0.94 which are very satisfactory.

V. Conclusion

The S-UMLDiff algorithm can help identify the differences between the two versions of the conceptual diagram and therefore the system analyst can use the differences report to verify if the changes that are present in the new diagram are correct and complete according to the change requirements of the new system to be developed. The analysis of semantic similarity of names and change of model element types enhances the algorithm and gives a more informative report of changes in the new version. However, the specified thresholds affect how S-UMLDiff classify changes, such as the case of the methods *deleteInfo* and *removeInfo* in Section III.B which may or may not be identified as a semantic match depending on how high the *Phrase Similarity Threshold* is. In addition, the system analyst's judgment and the score given by S-UMLDiff may sometimes be conflicting such as the case of the attributes *appId* and *appNo*. While the system analyst sees that they match and identifies them as *Rename*, the overall similarity between these two attributes are lower than the *Rename Threshold* and hence the algorithm identifies them as *Not Rename* and *appId* is reported as *Remove* and *appNo* as *Add*. The performance of the algorithm in terms of precision and recall, as a result, depends on these thresholds. As future work, S-UMLDiff and the supporting tool can be improved by visualizing the comparison results and even supporting change impact analysis.

TABLE III
Evaluation Results

Case#	$\lvert M_{actual} \rvert$	$\lvert M_{reported} \rvert$	$\lvert M_{actual} \cap M_{reported} \rvert$	precision	recall
1	11	11	10	0.91	0.91
2	15	15	15	1	1
3	8	8	8	1	1
4	28	34	25	0.74	0.89
5	20	20	19	0.95	0.95
6	37	37	36	0.97	0.97
7	24	30	22	0.73	0.92
8	24	30	22	0.73	0.92
9	20	22	18	0.82	0.9
10	34	34	33	0.97	0.97
			average	0.88	0.94

References

[1] Object Management Group. (2011, August). Unified Modeling Language [Online]. Available: http://www.omg.org/spec/UML/2.4.1/
[2] Z. Xing, "Supporting object-oriented evolutionary development by design evolution analysis," Doctoral dissertation, Department of Computing Science, University of Alberta, Canada, 2008.
[3] T. Pedersen. (2013, January). WordNet::Similarity [Online]. Available: http://wn-similarity.sourceforge.net/
[4] M. Girschick, "Difference detection and visualization in UML class diagrams," TU Darmstadt, Germany, Technical Report TUD-CS-2006-5, 2006.
[5] L. Auxepaules, D. Py, and T. Lemeunier, "A diagnosis method that matches class diagrams in a learning environment for object-oriented modeling," in *Proc. 8th IEEE Int. Conf. Advanced Learning Technologies (ICALT 2008)*, Santander, Cantabria, 2008, pp. 26-30.
[6] S. Sorlin, C. Solnon, J.-M. Jolion, "A generic graph distance measure based on multivalent matchings," *Applied Graph Theory in Computer Vision and Pattern Recognition, Studies in Computational Intelligence*, vol. 52, pp. 151-181, 2007.
[7] F. Giunchiglia, M. Yatskevich, and P. Shvaiko, "Semantic matching: algorithms and implementation," *J. Data Semantics*, vol. 9, pp. 1-38, 2007.
[8] Tigris.org. (2009). ArgoUML [Online]. Available: http://argouml.tigris.org/
[9] W. Gad and M. Kamel, "PH-SSBM: Phrase semantic similarity based model for document clustering," in *Proc. 2nd Int. Symp. Knowledge Acquisition and Modeling*, DC, 2009, pp. 197-200.
[10] C. Stein. (2012, September). Longest Common Subsequence [Online]. Available: http://www.columbia.edu/~cs2035/courses/csor4231.F11/lcs.pdf
[11] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Second edition. Essex: Addison Wesley, 2011.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)