Journal of Theoretical and Applied Information Technology
15th June 2018. Vol.96. No 11
© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

DETERMINING THE SIMILARITY OF UML-MODELS BY COMBINING DIFFERENT SOFTWARE PROPERTIES

[1]ALHASSAN ADAMU, [2*] WAN MOHD NAZMEE WAN ZAINON
[1]Department of Computer Science, Kano University of Science and Technology Wudil, Nigeria
[2]School of Computer Sciences, Universiti Sains Malaysia, Malaysia
E-mail: [1]*kofa062@gmail.com*, [2]*nazmee@usm.my*

ABSTRACT

One of the most important elements of every software system is its design architecture. Software design is a demanding task that requires lot of experience, expertise and knowledge of many different types of design alternatives. Each software engineers acquires more specific knowledge as he/she participate in a new project. Experienced engineers are very vital asset to Software Company, especially in a high competitive market environment; as such reusing knowledge of experienced engineers can save a lot of cost and time to the software company. UML models are de facto modelling language used by many software engineers during the software design stage, its receiving a widespread attention in the field of software reuse. It's not surprising, because of the benefits that can be reaped out during the reuse of early software design is numerous, and it can lead to reuse of all related work-products. There is considerable amount of works that takes place within the scope of UML models reuse, this paper presents an experimental results of different features of UML models that are used during the matching and retrieval of UML diagrams from repository.

Keywords: *UML Models, Similarity, Software Properties, Reuse.*

1. INTRODUCTION

Software reuse is the creation of software system using previously developed software rather than development from the scratch [1]. It helps to prevent the *reinvention of the wheel* during the software development. The benefit of software reuse includes accelerated software development, risk reduction process, effective use of specialists, reduction of development time, improvement of productivity and increase in the overall quality of software products [2]. However, these advantages do not come without any drawbacks. According to Salami and Ahmed [3], some of the challenges of software reuse include increased effort to create and maintain components library, effort to find and adapt reusable components, lack of tool supports and increase in maintenance cost.

According to Kotonya, Lock [4] every year, more than $5 billion worth of software projects are cancelled or abandoned worldwide. Many of these projects are dropped not because their software failed but because the project objectives and assumptions changed. Usually, the failed software projects are locked in potentially reusable software components. If we can find efficient ways to salvage and reuse these components, significant amount of the original investment can be recovered and new software can be developed rapidly at low-cost.

There are two types of software reuse: systematic and opportunistic [5]. In systematic reuse, software is particularly developed to be used in the future. This results in robust, well documented, and thoroughly tested artifacts. However, according to Salami and Ahmed [3], Keswani, Joshi [6] these types of reuse requires time, effort and additional cost of making components reusable. Meanwhile, many organizations are unwilling to sacrifice since there is no guarantee that such components can be reused in the future. However, in opportunistic reuse, developers come to the conclusions that a component is reusable when they realised that the previously developed component can be used in the new software products. However, according to Salami and Ahmed [3] the components might not be in their best form of reuse.

Software reuse can be carried out in four phases: representation, retrieval, adaptation, and incorporation [7]. During the representation phase, the fragment (i.e. query) of the software to be developed is presented. In the retrieval phase, the software components that are similar to the query with minimal adaptation cost are selected from the repository. During the adaptation, the components are modified to suite the need for the current software under development. Finally, in the incorporation

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

phase, the new software components are integrated back to the repository for future reuse.

Software systems consist of many artifacts that can be used to develop other software. These artifacts can range from the software requirement gathering down to software documentations, with source code reuse as the most practice type of reuse. However, reuse at the source code level represents a small fractions type of software reuse, because the reuse is delayed to occur at the later-stage of software development. The benefit of software reuse can be multiplied if it occurs at the early stage (such as software designs) of software development [8], because all the corresponding later stage artifacts can also be reused. However, the main challenges faced by software developers is the appropriate tools that can support the matching and retrieval of previous software artifacts from repository.

There are four stages that are involved in software reuse: representation, retrieval, adaptation and incorporation [9]. At the representation the initial draft of the software to be developed is presented as query to the reuse system. The software artifacts that are similar to the query with minimal adaptation cost are selected in the retrieval stage. During the adaptation, the retrieved artifacts are modified for future reuse. Finally, at the integration stage the new artifacts are stored back to repository. Among all the reuse stages retrieval plays a critical role [7]. It consists of two main activities: navigation and matching. The navigation determines the order in which artifacts are visited in the repository, while the matching defines the order in which artifacts are selected based on their similarity with the query draft. This paper focus on the matching based on the similarity between the software designs modelled with UML diagrams.

Similarity assessment of UML diagrams is the task that correspond to identifying the semantic correspondence between elements of two diagrams (e.g. class names). It is task that is error-prone, because these diagrams while representing similar software system functionalities are used independently by different software engineers, thus creating inconsistencies and design differences among the diagrams.

Most of the existing works in the literature software design reuse rely on the use on single type information contained in the diagrams, with some relying on the information contained inside the UML diagrams, for example the studies by Robles et. al [10] and Gomes and Leitão [11] which focus only class diagrams. Others focus on the structural representation of the diagrams and usually formulated as graph matching problems, such as the work of Salami and Ahmed [8], Park and Bae [7], and Assuncao and Vergillio [12]. This paper proposed an approach on how the similarity between UML diagrams can be computed by combining different software properties.

2. RELATED WORKS

There are several works in the literature that focus on the reuse of software design, with some relying on the information contained inside the UML diagrams, for example the studies by Robles et. al [10] and Gomes and Leitão [11]. Others focus on the structural representation of the diagrams and usually formulated as graph matching problems, such as the work of Salami and Ahmed [8], Park and Bae [7], and Assuncao and Vergillio [13]. Most of the existing work that consider the internal information of UML diagrams to compute the similarity are based on ontologies.

The ontology approaches rely on the semantic meaning of the objects in UML diagrams, for example the names of classes, attributes, and methods. This approach is particularly important when all the names of the objects appearing in the diagrams are written using Standard English words. However, if the object names in the class diagrams are not Standard English words, the ontology will break. Therefore, the similarity values returned by such approaches will be inaccurate. On the hand, the structural similarity is based on the relationship between the classifiers. Class diagrams are converted to equivalent graph representation in which the classes represent the nodes of the graph and the relationship between the classes represent the edges of the graph.

Significant research has been carried out on UML-based matching. For example, Ali and Du [14] used conceptual graph to aid the retrieval of software models. The similarity computation was based on the estimation of the conceptual distance between terms in the query and the terms in repository models.

On the other hand, Robinson and Woo [15] compute the similarity between sequence diagrams using SUBDUE [16] graph matching algorithm. Sequence diagrams were represented as conceptual graphs in which the object names in the sequence diagrams represents vertices, and the relationships between the diagrams (messages) represented the edges of the graph. The SUBDUE algorithm find the similarity between the graph by comparing the substructures of sequence diagrams in query and repository.

Srisura, Daengdej [17] proposed an approach of retrieving previous use case diagrams stored in repository using CBR. The retrieval method is based on two dimensions: use case and actor dimension, and relationships dimension. The use case and actor dimension consist of use case, actor, and system boundary components represented as text-based information. During retrieval, the words found in query and repository are extracted and formed a searched dictionary.

The similarity is computed as the average of the number of matched words found between query and repository use case diagrams. In the relationships dimension, the similarity is calculated based on three subcomponents: the relationship type, navigator, and multiplicity relationship. Each of the relationships is assigned a weighted value indicating the influence of the relationship in the diagram. Finally, the actual degree of similarity is returned by the CBR engine, and the appropriate diagrams were selected for reuse.

More research in this concern by Park and Bae [7] put forward two-stage framework to retrieve UML artifacts from repository. In the first stage the similarity between class diagrams was computed using structured mapping engine (SME). SME is analogical reasoning mapping technique which allows mapping of knowledge from one domain to another by considering the relational communalities between objects in the domain regardless of the objects involved in the relationships. The subset of the repository UML projects was selected for subsequent comparison using class diagram. In the second stage, sequence diagrams in the shortlisted models were converted to message-order-graph (MOOGs), where nodes denote the location where events occur (message send or received) in sequence diagrams and the edges denote the flow of events between objects and the flow of time inside each object. The similarity between two MOOGs was computed based on the number of nodes and edges in each of the graph using graph matching algorithm.

Paydar and Kahani [18] suggested a semi-automatic approach to adapt UML activity diagrams to create new use case diagrams. The information regarding use case diagrams, activity diagrams and class diagrams are stored in a model repository. Consequently, the similarity of two use cases was computed based on their semantic similarity. The semantic similarity was computed in two aspects: the similarity of the sole use cases and the similarity of the context in which the use case exists. The measure of the semantic similarity was based on WordNet. Finally, the semantic similarity of two use cases was computed as the weighted sum of their similarity values.

In [19] the similarity between class diagrams is computed from their graph representation, in which class diagrams are converted to a weighted directed graph, where class names represent the nodes of the graph and the relationship between the classes represents the edges of the graph. The similarity between two class diagrams is computed from the adjacency matrix of their graph representation with the aid of GA. The adjacency matrix contained the type of relationships between the classifiers.

In [13] particle swamp optimization algorithm was used to aid the retrieval of UML class diagrams from repository. The similarity between two class diagrams is computed as a n aggregation of two similarity measures: (i) name similarity and (ii) relationship similarity. The name similarity is computed using Levenshtein Distance [20] as the measures of the number of characters in a strings required to change to obtain another string. The relationship similarity is computed based on the relationship between classifiers in a class diagram.

In [21] a method of computing the similarity between query and repository models is presented. The similarity between class diagrams is computed using three type of UML diagrams information: (i) lexical naming information (shallow), (ii) internal information, and (iii) neighborhood information. The shallow lexical information is used to compute the similarity between entities names in UML class diagram. The internal information is used to compute the similarity of the internal properties of classes (i.e. attributes) and the behaviors (i.e. operations).

The neighborhood information is used to compute the similarity of the structural relationships of class diagrams. The similarity between concepts in class diagrams are computed based on their semantic similarity (e.g. synonyms, hyponyms) according to the WordNet is–a hierarchy of concepts. The neighborhood information is compute from the graphical representation of class diagrams. Finally, greedy algorithm is applied to find the correspondence between the elements as an aggregation of the similarity measures.

Furthermore, a similar approach was reported by [22] where sequence diagrams were converted to a directed graph, the similarity between the graphs was determined with the aid of genetic algorithm (GA). The GA helped to terminate the searching process in order to avoid exhaustive comparison. The termination criteria were based on three conditions: first, if the fitness value reached 0, it indicated the maximum similarity between class diagrams, second, if the maximum number of iteration reached, or if the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

fitness function did not improve within a given number of iterations.

In more recent studies by [1] state machine diagrams retrieval approach was presented. State machine diagrams were represented by finite state machine diagram in which i) every states in the state machine diagrams represents states in the finite state machine, ii) the transition between one state to another in state machine represents the transition in finite state machine. The similarity between state machines diagrams is computed by means of similarity function table containing the differences between the various types of relationship in UML state machine diagram.

Different to the existing works, this paper presents an approach of accessing the similarity between UML-models by combining different software properties. Typically for each software system there is a set of UML models that describe it's structural, functional, and behavioral perspectives [21]. Our focus on this paper is on two different diagrams namely class and state machine diagrams.

3. RETRIEVAL APPROACH

Retrieval involves the process of matching query and repository diagrams focusing on the most useful related to the problem at hand. A similarity measure has to be applied to allow retrieving the most similar diagrams. The retrieved diagrams provide a solution to the new problem at hand. Fig. 1 shows the retrieval engine cycle: it consists of old problems and their solutions stored in the repository. The repository is a library system for storing and managing of software components for building business applications. It supports the storing, registration and management of all software artifacts produce during software development lifecycle, and support the reuse of those components. It can contain different information, depending on the scope of the system [23].

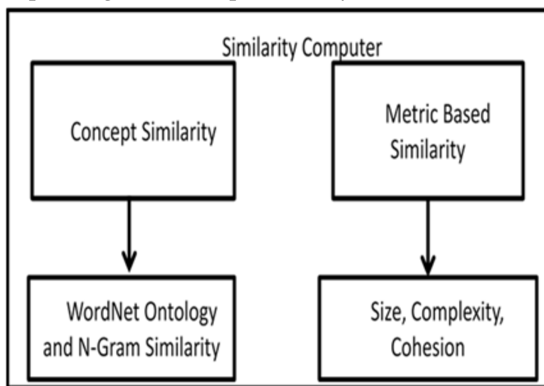

Fig. 1: Retrieval System

In similarity computation, query and repository diagrams are retrieved based on measuring their similarity. The usefulness of a diagram is estimated based on the presence or absence of similar features between the query and repository. The similarity is access through numeric computation and reflected as a single value; for example weighted sum, which shows all aspect of the similarity. There are three similarity metrics to be used by the retrieval engine. These metrics are Concept Similarity computation (CSim), Functional Similarity Computation (FSim) and Metric Based Similarity Computation (MBSim) as shown in Figure 2.

Concept similarity computation is performed by comparing the concept name appearing in both query and repository models with the aid of WordNet ontology. WordNet is built around the concept of synset. Synset is concept represented by one or more words. One words can be used to represents the meaning of the same synset, for example a word mouse have two meanings, it can refers to computer mouse or rat; a words that can be used to represents one synset are called synonyms while words with more than one meaning are referred to as polysemous.

If the concept appearing in query and repository are not valid English word, the similarity computation can break since WordNet ontology is centered on the use of valid English words. In this N-Gram similarity is applied to compute the similarity based on the number identical substrings of length n contained in both strings. The second approach of similarity computation is the Metric Based. Computes the similarity between query and repository by comparing the metric values of both query and repository diagrams. It is expected the corresponding metric for similar software should not differ significantly.

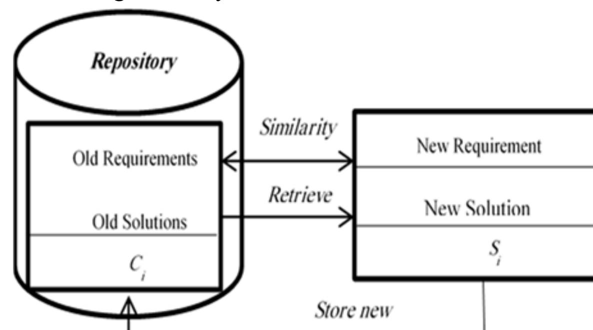

Figure 2: Similarity Computation Approach

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Software engineers are often faced with many problems in finding perfect solutions. These solutions are either impossible or impractical to achieve. Software engineers are left with the options of near optimal solutions or solutions that fall within a specified acceptable tolerance. Precisely, these factors make robust metaheuristics search-based optimization technique readily applicable [24]. Taking into consideration when exploring large repository, the search space can be exponential since huge number of candidates solutions need to be analyzed, accordingly finding a mapping that produces optimal similarity of UML artifacts represents an NP-hard problem. These limitations motivated some authors to use heuristics search techniques, particular Genetic Algorithm [19] and Particle Swamp Algorithm [8] to properly deal with UML artifacts retrieval problem. Different to the existing approaches, this paper proposed the use Harmony Search Algorithm to aid the retrieval of similar diagrams from repository.

Harmony Search algorithm (HS) was developed by Geem, Kim and Loganathan [25] in 2001. The algorithm mimics the behaviors of music improvisation process. Harmony search had been applied in a wide range of optimization problem, and has proved to have several advantages over the traditional optimization techniques. The general procedures of harmony search algorithm is describe as follows:

Step1: Create and randomly initialize a harmony memory (HM) with size HMS.
Step2: Improvise a new harmony from the HM.
Step3: Update the HM. If the new harmony is better than the worst harmony in the HM, include the new harmony into the HM, and exclude the worst harmony from the HM.
Step4: Repeat Steps 2 and 3 until the maximum number of iterations is reached.

Harmony memory is a set of solution vectors [26], it is similar to population in genetic algorithm. It is governed by three distinct rules: i) the harmony memory size (HMS), or the number of solution vectors in the harmony memory; harmony consideration rate (HMCR); pitch adjustment rate (PAR); and finally the number of improvisation or the algorithm stopping criteria [27]. Figure 3 shows how Harmony search algorithm is used in the retrieval of similar diagrams from repository.

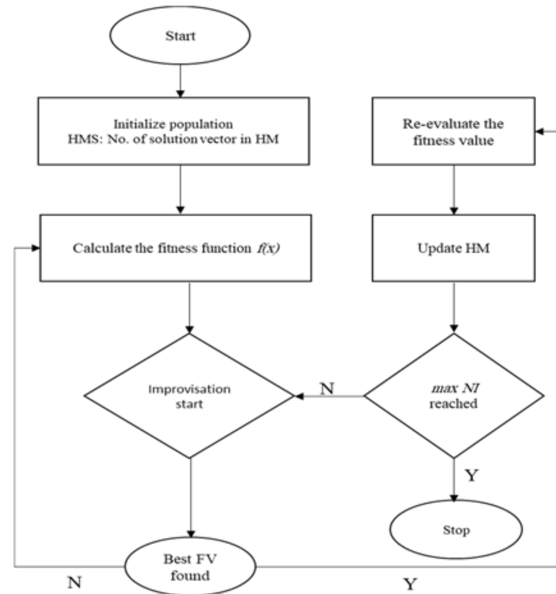

Figure 3: Retrieval with Harmony Search Algorithm

At the initial stage, a new harmony vector or population is generated based on the three rules: memory consideration rate, pitch adjustment rate and random selection. This procedure is called improvisation in harmony search algorithm. The initial vectors are generated randomly from the old software designs in the repository. Once the initial population is formed the fitness value of each solution vector in the HM is computed and the best values are determined. The improvisation continues until the stopping criteria is reached. If the best value is found the algorithm will re-evaluate the fitness function and determined the new best fitness values and then update Harmony Memory, otherwise the algorithm will run until the maximum number of improvisation reached.

4. SIMILARITY MATCHING

Similarity estimated the present or absent of similar features between query and repository diagrams. The similarity is computed through a numeric computation and reflected a single value indicating how similar the query is with the existing software projects in the repository, the value always lies between 0 and 1. There are three similarity measures used during the Multiview similarity assessment: structural similarity, functional similarity and behavioral similarity. Each of this similarity is discuss in the subsequent subsections.

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4.1 Structural Similarity

Class diagram depicted the structure of a system by showing the system's classes and the relationships between them. This section discusses the metrics for computing the similarity between different software systems by comparing the system UML class diagrams. Class diagrams consist of two different properties:

i) The structural relationship between the classifiers in class diagrams and
ii) The conceptual elements within the class diagrams classifiers.

Therefore, the similarity of class diagrams can be computed as the aggregation of these two elements. For the structural relationship the similarity between class diagrams is computed as the mapping of one type of relationship to another type of relationship in different class diagrams. For example what is the cost of mapping association relationship to an aggregation relationship? Interested readers may refer to [13, 19] for more details. On the other hand, the conceptual similarity is inspired from [13] in which the similarity of the internal information in the classifiers was computed as the edit distance between the classifier's elements Edit Distance is the minimum number of edits required to transform one string into another string [28]. It had several applications in the areas of bioinformatics such as DNA or protein alignment, file comparison, gas chromatography and speech recognition [29]. The similarity between two class diagrams is computed based on the minimum (characters) of edits required to transform one concept in one diagram to another.

Let C_1 and C_2 be two class diagrams having number of classifiers of size of $|S_1|$ and $|S_2|$ respectively, the similarity measure of two class diagrams can obtain from the number of matching classifiers. We defined a mapping (R_1,R_2) from one class diagram to another if the relationship between classifiers in class diagram A mapped to the type of relationship in classifiers in class diagram B as shown in Eq. (1) as follows:

$$Mapping(C_1,C_2) = \begin{cases} R_1 = R_2 \ \forall R_i \in S_i : CA_{i_{src}} = CB_{j_{src}} \text{ and } CA_{i_{dest}} = CB_{j_{dest}} \\ R_i \neq R_j & \text{otherwise} \end{cases} \quad 1$$

$CA,B_{i,j}$ are number of classifiers in class diagram A and B respectively. R_1 and R_2 are the types of relationships contained in class diagrams A and B.

The similarity between two class diagrams is computed as the aggregation of the structural and the internal information between the classifiers in class diagrams using Eq. (2) as follows:

$$Sim(C_1,C_2) = Sim(map(R_{c1},R_{c2}) * w_1) + Sim((ED(C_{I1}, C_{I2}) * w_2 \quad 2$$

Where C_1 , C_2 are two class diagrams, map(R_{c1} , R_{c2}) are the mapping of the relationship in one class diagram to another, ED(C_{I1} , C_{I2}) is the similarity of internal information contained in class diagrams, and w_1 and w_2 are weight factors that determined the relative importance of structural and internal information of class diagrams.

4.2 Behavioral Similarity

The behaviour of a software system is manifested in state machine diagram. State machine is a behavioural diagram that portrays the states an object goes through during its life time in response to events, together with the responses to those events by the object. The similarity between two state machine diagrams can be computed from their graphical representation. The approach of accessing the similarity of groups of state machine diagrams is also discussed since software system are hardly modelled using single state machine diagram.

The similarity of two state machine diagrams can be computed from their given transition matrix representations. Let S_1 and S_2 be two state machine diagrams whose degree of similarity is to be determined, and let Tm be the transition matrix that contained all the type of relationship between one state and another in state machine diagram. The similarity of two state machines diagrams can be computed using Eq. (3).

$$Sim(S_1,S_1) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} Tm(i,j)}{\max(Tm(i,j)} \quad 3$$

Where Tm_i and Tm_j are the transition matrix of S_1 and S_2, *max* are functions that return maximum values of two of its arguments, *i,j* are the number of rows and columns in Tm. Interested readers may refers to our earlier work [1] for more detailed.

4.3 Aggregation of Two Similarity Methods

Software system is commonly modelled from different perspectives using different UML diagrams. These diagrams represent the different views of software. Rather than relying on single views, this section discussed the way of accessing the similarity of software system from multiple views by

Journal of Theoretical and Applied Information Technology
15th June 2018. Vol.96. No 11
© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

aggregating individual similarity measures into one single similarity measure

Since software designs are modelled using more than one type of UML diagrams, assessing the similarity of software designs using multiple diagrams may produce better similarity value compares to using only one single diagram. This paper combine two type of UML diagrams to determine the similarity between software designs. The approach is based on the aggregation of structural and behavioral views of software systems using class and state machine diagrams respectively. The similarity is computed using Eq. (4) as follows:

$$\boldsymbol{Aggregation}(\boldsymbol{Struc}, \boldsymbol{Behv}) = \sum \boldsymbol{Sim}(\boldsymbol{sim}(\boldsymbol{c1}, \boldsymbol{c2}) * \boldsymbol{w1}, \boldsymbol{sim}(\boldsymbol{s1}, \boldsymbol{s2}) * \boldsymbol{w2}) \quad (4)$$

Where sim($c1$,$c2$) is the similarity values obtained from structural similarity computation and sim(s1,s2) is the similarity value obtained from behavioral similarity computation, w1 and w2 are the weight factor that determined relative important of the similarity assessment method.

5. EXPERIMENT

A repository of six projects containing class diagrams and state machine diagrams as shown in Table 1 has been created. The projects were created by randomly adding, changing, and/or deleting class diagrams relationship and state machines transitions. For example, generalization relationship can be change to composition relationship. The projects were obtained from undergraduate student projects. The original project before alterations are used as the queries.

Table 1: Description of Query used for experiment

Query	Description	#class diagrams	#state machine diagrams
P_1	Bank System	1	1
P_2	Online Booking system	1	3
P_3	Traffic Management System	1	4
P_4	Student Course registration system	1	2

6. RESULTS

The matching quality was measured using Mean Average Precision (MAP), a measured commonly used to evaluate information retrieval system. Average precision (AP) for a given query was obtained using precision values calculated at each point whenever a new project was retrieved (i.e. precision = 0 for each of the relevant project that is not retrieved). The Mean Average Precision for a set of query was the mean of the AP scores for each query, also referred as mean precision at seen relevant projects [18]. The formula is given in Eq. (5).

$$MAP = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{Q_j}\sum_{i=1}^{Q_j}P(rel = i) \quad (5)$$

N is the number of queries, Q_j is the number of relevant documents for query j and P(rel=i) is the precision at the ith relevant document.

Additionally, the time to search repository by each method is measured as the time taken to return similar software designs with the query. The mean average precision and average time required to search the repository is shown in Figure 4 and Figure 5 respectively. Figure 4 shows the comparison of the percentage of time when each of the method return better similarity values.

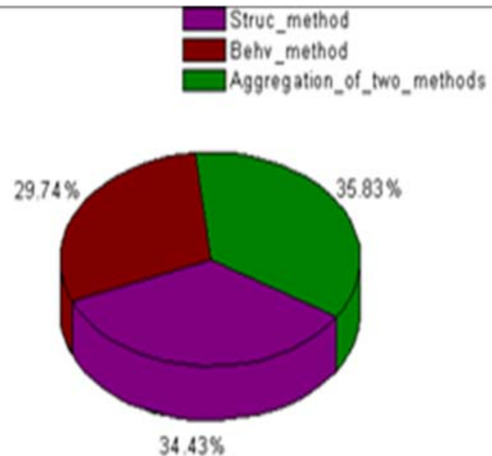

Figure 4: Number of times when each Method produce better similarity values (MAP)

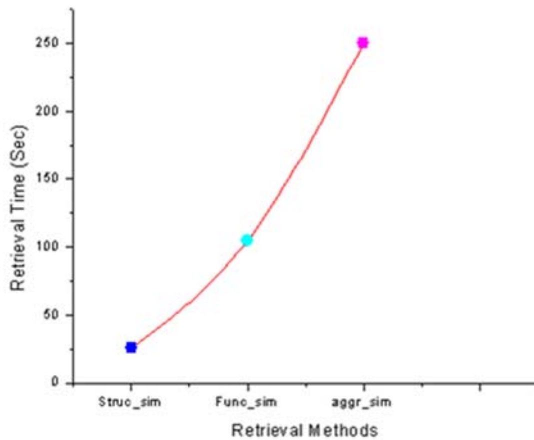

Fig 5: Time to search Repository by each Method

It can be examined that structural similarity assessment methods produce the best MAP compared to behavioral similarity assessment method with 34.43% as against 29.74%. Moreover, the aggregation of the two methods produce better MAP as compared to the single methods with 35.83% of time when it produce better similarity values. It is not surprising that the aggregation of the approach produces better similarity as against the single methods, this is because both cases of the structural and behavioral are considered during the similarity assessment.

However, this come with a price as can be observed from Figure 4 that the aggregation of the two method required more time to search the repository compared to the single method. The structural method required less retrieval time compared to the other methods, this is because only the classifiers are involved in this similarity assessment method, which requires a small search space compared to other method. A method of computing the similarity between software projects based on the metric values (e.g. size) was presented. The metric based similarity of two UML artefacts was computed by comparing the metric values in query and repository projects. The set of metric values of UML artefacts in query and that in repository were presented by dimensional feature vector space. Each of the dimension held the information regarding particular metric. The metric similarity of two UML artefacts was calculated as the Euclidean distance between their feature vectors. These metrics were particularly used during the pre-filtering of repository projects.

Behavioral similarity assessment method requires more time as compared to the structural method. This is expected because it is normal to represent one class diagrams with many number of state machine to represent the behavior of the objects involved in a class diagrams, as result of this the search space when state machine is used become very large as compare to when class diagram is used.

7. CONCLUSION

The benefits of software reuse can be maximized if it is carried out at the early-stage. This paper discussed the retrieval of software projects from repository based on the degree of similarity of their UML diagrams. Two important issues were addressed:

1. Software systems are usually described from multiple perspectives using UML diagrams, which results to inconsistencies during retrieval if not properly handled.
2. A repository usually contained voluminous software projects with many diagram entities, thus exhaustive mapping of this entities during similarity computation is computationally demanding.

This paper presents an approach for computing the similarity between UML diagrams, the aim of this paper is to compare the effect of different UML diagrams properties in accessing the similarity between software designs. Several similarity measures are being discussed and experimental results has been presented.

The results of the experiment show that the use of aggregation produce better similarity values compared to other methods. Therefore, it is recommended to use this method especially when the repository size is not large. On the other hand, structural similarity produce better similarity as compared to behavioral method and it requires less retrieval time compared to the other method, therefore it is recommended to use this method especially when the size of the repository is large since it return the similarity values faster than the other methods.

Finally, this paper concentrate only on two types of diagrams class and state machine diagrams representing the structural and behavioral view of the software system, in the future we intend to include other UML diagrams such sequence and deployment diagrams in order improve the MAP of the aggregation similarity assessment method. Furthermore, in the future we plan to develop a tool in order to assist the reuser to retrieved old software designs from repository and integrate into new software development. At present, the proposed approach is implemented as prototype for evaluation purpose. Henceforward, a case tool should be developed and integrated into UML diagrams tools for effective reuse of UML diagrams.

Next, the behavioral similarity assessment does not take into account the names of states, events, and guards conditions at this moment. In the future, all of these should be considered to improve the retrieval quality of state machines diagrams.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Higher Education of Malaysia, under the Fundamental Research Grant Scheme (FRGS: 203/ PKOMP/6711533).

REFERENCES

1. Frakes, W.B. and K. Kyo, *Software reuse research: status and future.* Software Engineering, IEEE Transactions on, 2005. **31**(7): p. 529-536.
2. Al-Badareen, A.B., et al. *Reusable software components framework.* in *European Conference of Computer Science (ECCS 2011).* 2010.
3. Salami, H.O. and M.A. Ahmed, *UML artifacts reuse: state of the art. arXiv preprint arXiv:1402.0157, 2014.*
4. *Kotonya, G., S. Lock, and J. Mariani, Scrapheap software development: lessons from an experiment on opportunistic reuse. IEEE software, 2011. 28(2): p. 68-74.*
5. *Kulkarni, N. Systematically selecting a software module during opportunistic reuse.* in *Proceedings of the 2013 International Conference on Software Engineering. 2013. IEEE Press.*
6. *Keswani, R., S. Joshi, and A. Jatain. Software Reuse in Practice. in Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on. 2014. IEEE.*
7. *Park, W.-J. and D.-H. Bae, A two-stage framework for UML specification matching. Information and Software Technology, 2011. 53(3): p. 230-244.*
8. *Salami, H.O. and M. Ahmed. Class Diagram Retrieval Using Genetic Algorithm. in Machine Learning and Applications (ICMLA), 2013 12th International Conference on*. 2013. IEEE.
9. Salami, H.O. and M. Ahmed, *A framework for reuse of multi-view UML artifacts. arXiv preprint arXiv:1402.0160, 2014.*
10. *Robles, K., et al., Towards an ontology-based retrieval of UML Class Diagrams. Information and Software Technology, 2012. 54(1): p. 72-86.*
11. *Gomes, P. and A. Leitão. A tool for management and reuse of software design knowledge. in International Conference on Knowledge Engineering and Knowledge Management. 2006. Springer.*
12. *Wesley Klewerton Guez Assunc, S.R.V., Class Diagram Retrieval with Particle Swarm Optimization, in 25th International Conference on Software Engineering and knowledge Engineering (SEKE 2013)*. 2013. p. 632-637.
13. Assunçao, W.K.G. and S.R. Vergilio. *Class Diagram Retrieval with Particle Swarm Optimization*. in *The 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013)*. 2013.
14. Ali, F.M. and W. Du, Toward reuse of object-oriented software design models. Information and software technology, 2004. 46(8): p. 499-517.
15. Robinson, W.N. and H.G. Woo, Finding reusable UML sequence diagrams automatically. Software, IEEE, 2004. 21(5): p. 60-67.
16. Jonyer, I., D.J. Cook, and L.B. Holder, Graph-based hierarchical conceptual clustering. The Journal of Machine Learning Research, 2002. 2: p. 19-43.
17. Srisura, B., et al., Retrieving use case diagram with case-based reasoning approach. J. Theor. Appl. Inf. Technol, 2010. 19(2): p. 68-78.
18. Paydar, S. and M. Kahani, A semi-automated approach to adapt activity diagrams for new use cases. Information and Software Technology, 2015. 57: p. 543-570.
19. Salami, H.O. and M.A. Ahmed. A Framework for Class Diagram Retrieval Using Genetic Algorithm. in SEKE. 2012.
20. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. in Soviet physics doklady. 1966.
21. Al-Khiaty, M.A.-R. and M. Ahmed. *Similarity assessment of UML class diagrams using a greedy algorithm.* in *Computer Science and Engineering Conference (ICSEC), 2014 International*. 2014. IEEE.
22. Salami, H.O. and M. Ahmed. *Retrieving sequence dia*grams using genetic algorithm. in Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on. 2014. IEEE.
23. Subedha, V. and S. Sridhar, Design of a Conceptual Reference Framework for Reusable Software Components based on Context Level. IJCSI International Journal of Computer Science Issues, 2012. 9(1).
24. Harman, M. and B.F. Jones, Search-based software engineering. Information and Software

Technology, 2001. 43(14): p. 833-839.
25. Geem, Z.W., J.H. Kim, and G.V. Loganathan, A new heuristic optimization algorithm: harmony search. simulation, 2001. 76(2): p. 60-68.
26. Wang, C.-M. and Y.-F. Huang, Self-adaptive harmony s*earch algorithm for optimization.* Expert Systems with Applications, 2010. **37**(4): p. 2826-2837.
27. Mahdavi, M., M. Fesanghary, and E. Damangir, *An* improved *harmony search algorithm for solving optimization problems.* Applied mathematics and computation, 2007. **188**(2): p. 1567-1579.
28. Herman, D., Asset Reuse of Images From a Repository. 2014, Walden University.
29. Begum, A., A Greedy Approach For Computing Longest Common Subsequences. Journal of Prime Research in Mathematics, 2008. **4**: p. 165-170.