International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

Similarity Detection in Collaborative Development of Object-Oriented Formal Specifications

Fathi Taibi, Fouad Mohammed Abbou, Md. Jahangir Alam

***Abstract*—**The complexity of today's software systems makes collaborative development necessary to accomplish tasks. Frameworks are necessary to allow developers perform their tasks independently yet collaboratively. Similarity detection is one of the major issues to consider when developing such frameworks. It allows developers to mine existing repositories when developing their own views of a software artifact, and it is necessary for identifying the correspondences between the views to allow merging them and checking their consistency. Due to the importance of the requirements specification stage in software development, this paper proposes a framework for collaborative development of Object-Oriented formal specifications along with a similarity detection approach to support the creation, merging and consistency checking of specifications. The paper also explores the impact of using additional concepts on improving the matching results. Finally, the proposed approach is empirically evaluated.

***Keywords*—**Collaborative Development, Formal methods, Object-Oriented, Similarity detection

I. Introduction

SIMILARITY detection is an important issue in several computer-related fields. They include (among others) information/software retrieval [11], software reuse and evolution [2], model management [12], collaborative design/development [8], and plagiarism detection [15]. A similarity detection approach should identify the mappings that exist between the elements of some objects of interest (views) or between a query and a repository of objects. These mappings are generally calculated based on syntactic similarity as well as on some other aspects, such as behavioral or structural similarity. Similarity results are useful when they can identify precisely most of mapping that exist between the views.

Syntactic similarity is computed by comparing strings. Algorithms such as Longest Common Substring (LCS) [3] and N-gram [7] could be used for this purpose. Syntactic similarity helps determining the early correspondences based on which matching is performed and cannot alone be the basis of similarity detection. Behavioral or structural similarity is computed based on the hierarchical relationships or associations that exist among the objects of the views of interest as well as the structures of these objects and the elements they contain. Here data structure such as trees are used to represent these links, and the computation is based on ancestor, descendent, neighbors, and content similarities.

An overall similarity comprising both syntactic and structural (or behavioral) similarities could be used to determine the mappings that exist between the views of interest based on a chosen threshold. The threshold, which is a number between zero and one, represents the strictness of the mapping approach. High threshold may lead to low number of false positives (good precision), however, it may lead to many missed matches (poor recall). Low threshold may lead to low number of false negatives (good recall); however, it may lead to many false matches (poor precision). A good matching approach must be efficient (in time and space) and should provide good precision combined with good recall. However, this is a very challenging problem in practice. Yet, a matching approach that provides balanced results (precision vs. recall) is possible to achieve provided it adopts the right concepts, and uses the right techniques.

The complexity of today's software systems makes collaborative development necessary to accomplish tasks. During requirements engineering, several developers participate to create a software specification. Each developer creates a partial specification of the software under development based on a particular perspective (view), the different specifications (views) have areas of overlap, and their combination (merging) results in a complete software specification. Similarly, during coding, several developers' code different aspects (packages or modules) of a software system and the latter is obtained by integrating these programs. Frameworks are necessary to allow developers perform their tasks independently yet collaboratively. Similarity detection is one of the major issues to consider when developing such frameworks. It allows developers to mine existing repositories when developing their own views of a software artifact, which facilitates reuse. Moreover, it

F. Taibi is with the Faculty of Information Technology, University of Tun Abdul Razak, Kelana Jaya (47301), Selangor, Malaysia. Phone: 603-78092068. Fax: 603-78802404. Email: taibi@unitar.edu.my.

Dr. F. M. Abbou is with Alcatel-Lucent. He is attached to the Faculty of Engineering, Multimedia University, Cyberjaya (63100), Selangor, Malaysia. Email: fouad@mmu.edu.my.

Dr. M. J. Alam is with the Faculty of Information Technology, Multimedia University, Cyberjaya (63100), Selangor, Malaysia. Email: md.jahangir.alam@mmu.edu.my.

allows identifying the correspondences between the views to allow merging [10] them and checking their consistency [14]. Based on a good matching approach, merging the partial views helps obtaining the intended result from a particular development activity. Merging could even be used solely as a consistency checking technique [16]. When used at an early stage, such as during the development of software requirements, merging could help in identifying and resolving conflicts that cost higher to identify (and resolve) during later stages of software development, which improves software quality and reduces development cost.

Merging requirements specified informally (by textual or graphical means) is tremendously difficult and error prone due to the ambiguous nature of natural languages and the notations used. Formal methods offer a better alternative because of their precise and accurate nature.

In this paper, a framework for collaborative development of Object-Oriented (OO) formal specifications is proposed along with a similarity detection approach to support the creation as well as the merging of the specifications. The proposed approach incorporates heuristics for both syntactic and structural similarity.

In the following sections, the proposed framework and its components are discussed first. This is followed by discussing similarity detection and proposing a new matching approach for the proposed framework. After that, the impact of using additional concepts on improving the matching results is exploited. The matching approach is then empirically evaluated. This is followed by discussing related work, and the final section concludes the paper and discusses future work.

II. Collaborative Development of Object-Oriented Formal Specifications

Specifying software requirements requires the collaboration of several developers. The requirements themselves are derived through several stakeholders with different views (or perspectives) about the software under development. Because of the ambiguous (and sometimes misleading) nature of informal requirements, it is tremendously difficult to process them to perform critical tasks such as merging them and checking their consistency. Formal methods offer a better alternative in specifying software requirements because of their precise and accurate nature, which makes it possible for automatic verification through model checking. OO formal specifications have a double advantage as they combine the strengths of formal and OO methods. Thus, reuse is possible because of the OO nature of the developed specifications. The formal specification language Object-Z [17] is an OO extension of the well-established formal specification language Z [18]. Thus, making it a good candidate to be used during the collaborative development of software specifications.

Frameworks need to be developed to support collaborative work not only at the requirement stage, but also at all stages of software development. Because of the importance of the requirements stage, and the lack of focus (from the intended scope of this work) by current research on it, a complete framework to support the collaborative development of OO formal specifications is proposed. The framework's architecture is shown in figure 1.

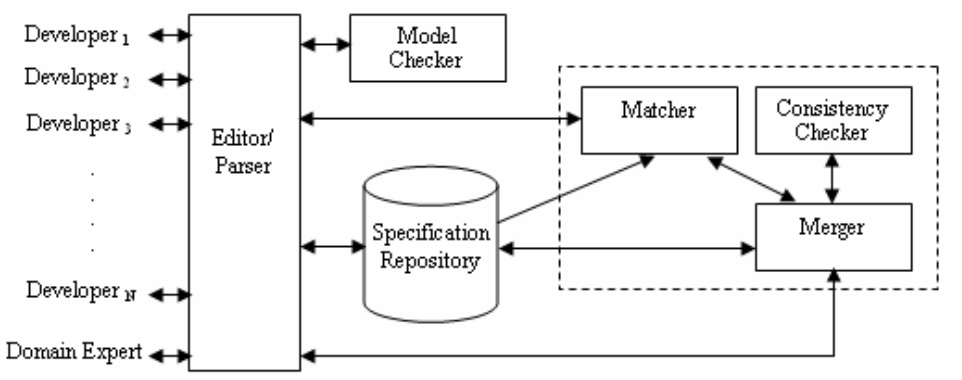

Fig. 1 Architecture of the proposed framework

The framework is based around three main approaches: matching, merging, and consistency checking. The matching approach, referred to as *Matcher* module in the framework, is the focus of this paper. It is responsible of detecting the similarities between different specification views. In addition, it helps mining a *Specification Repository* to support reuse, and the creation of landmarks between specification views. The merging and consistency checking approaches, referred to as *Merger* and *Consistency Checker* modules, are beyond the scope of this paper. They are responsible of combining specification views and ensuring that the merging result is consistent from an OO perspective.

In the framework, the use of the following operations is proposed:

1) Each developer creates an OO formal specification representing a particular view of the developed software. Object-Z could be used as specification language.
2) While creating their specifications, developers could mine a repository containing other specification views of the software system under development. They could edit, modify, and make the necessary additions based on their respective perspectives.
3) Based on mining results, developers could create landmarks between the classes (and the elements) of their views and those in the *Specification Repository*.
4) Individual views could be model checked before being saved into the *Specification Repository* to ensure the self-consistency of each view. For Object-Z specification, model checkers such as FDR [5] could be used to accomplish this task. Specification views of the same software system are stored together under the same package.
5) The *Matcher* module is responsible for identifying similarities between specifications. It could be used to mine the *Specification Repository* (as in 2). In addition, it provides a complete list of correspondences between the specification views that is used by the *Merger* and *Consistency Checker* modules. The matching results may be adjusted by a domain expert.
6) The *Merger* module is responsible for combining the specification views using the results of the *Matcher* module. It also ensures that the resulting specification is consistent by calling a *Consistency Checker* module. The merging result could be saved back in the *Specification Repository*. Merging is triggered by a domain expert by selecting the specification views involved.
7) The *Consistency Checker* module enforces structural consistency rules for OO specifications since each specification view is verified using a model checker (as in 4)
8) The *Editor/Parser* module provides an interface for specification creation, editing, and syntax checking. It also handles the interaction with the *Model Checker*, the *Matcher*, the *Merger*, and the *Specification Repository*.

Similarity detection in such frameworks is the focus of this work and the rest of the paper solely discusses the proposed matching approach. As a motivation example, consider the following Object-Z classes that are taken from two views (comprising 9 classes) of a specification of an online purchase system.

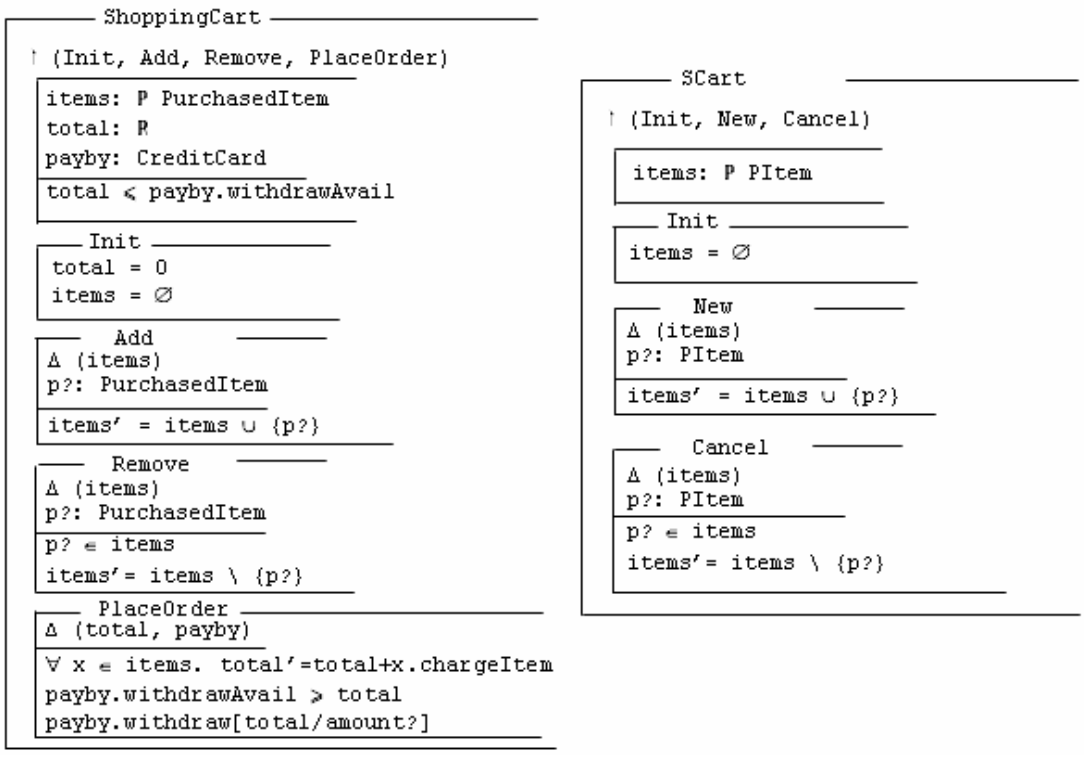

Fig. 2 Two views of an Object-Z class

In the class *ShoppingCart*, only the *Init* schema and the operations (*Add*, *Remove*, and *PlaceOrder*) are visible outside the class. *ShoppingCart* composes two other classes (*PurchasedItem* and *CreditCard)* by using them in defining its attributes. The class' invariant states that the total *amount* of purchased items of the shopping cart should be less or equal than the *withdrawal* amount available from the credit card used in the transaction. Initially, the shopping cart is empty

and its amount is null. The operation *Add* adds an item to the shopping cart. The operation *Remove* removes an item from the shopping cart. Finally, the operation *PlaceOrder* computes the total amount of all items included in the shopping cart and processes payment by calling *withdraw* operation of the class *CreditCard*. The class *SCart* represents another view of the class *ShoppingCart* taken from another specification created by a different developer. For esthetic reasons, the two full specifications (around 5 pages) cannot be included here; however, the following figure gives a preview of the classes involved and their relationships.

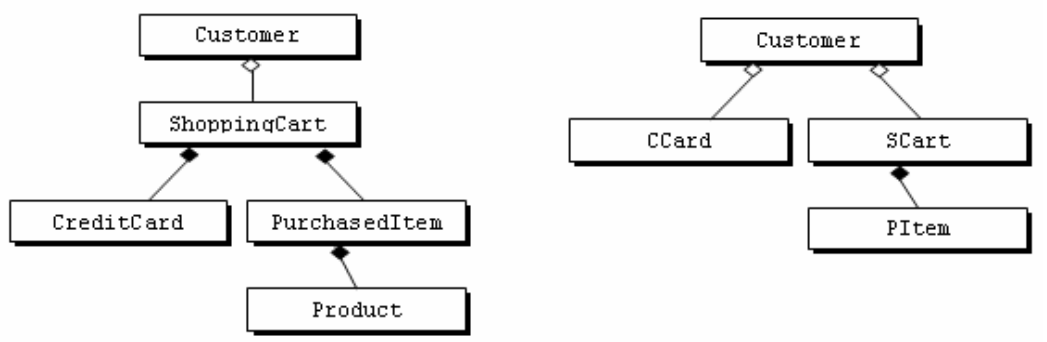

Fig. 3 The classes and their relationships of two views of the same system

The class *Customer* in both views is an obvious match. It is not the case of the other classes. From a syntactic similarity perspective, *ShoppingCart* could weakly be matched to either *CCard* or *SCart*. Same thing applies to the other classes, i.e. *CreditCard*, *PurchasedItem*, *Product* and *PItem*. Precisely matching the classes and relationships of the above views is a challenging problem. However, the following sections will show how this could be done with a relatively good precision.

III. A PROPOSED MATCHING APPROACH

Given two OO formal specifications S_1 and S_2, the matching approach computes an overall similarity metric between all the classes of S_1 and those of S_2. The matches are identified based on a chosen threshold. Figure 4 shows the matching algorithm.

```
Match
Input: 2 specifications S1=<A1,…,An>, S2=<B1,…,Bm>, and a threshold T
Output: A correspondence relation R storing the positive matches
between S1 and S2
1.   R=∅
2.   for all classes Ai of S1
3.    for all the classes Bj of S2
4.     x= overallSimilarity(Ai, Bj)
5.     if x≥T then
6.      R=R ∪ {(Ai, Bj)↦ x)
7.     else (Remove all elements from R associated with (Ai, Bj))
8.     end if
9.    end for
10.  end for
11.  return R
```

Fig. 4 The matching algorithm

Match makes a call to an overall similarity function *(overallSimilarity)* that return a value between 0 and 1 that combines both syntactic and structural similarities. The call is made between all the classes of the two specifications (line 4). The classes whose overall similarity is bigger or equal to a chosen threshold (line 5) are added to the correspondence relation (line 6). The latter will be an input to the merging process. The higher the threshold is, the stricter the similarity requirement is. The overall similarity between two classes could be used back to enhance the similarities between their elements. This concept is further discussed in section 4.

After a match is identified and added to the correspondence relation R, the class B_j is not discarded from the next round of comparison. Non-removal of a class B_j with a confirmed match in S_1 is motivated by the fact that it is up to the merging process (or a domain expert) to normalize the correspondence relation by choosing the correct match among the correspondences available in case of multiple matches for a given class (or element).

In addition to the matched classes, the correspondence

relation R stores the correspondences between the elements (such as attributes, operations and relationships) of the matched classes processed during the computation of the *overallSimilarity*. In case the overall similarity between two classes is below the threshold, all the elements added to R during the computation of the *overallSimilarity* (i.e. associated with the current classes (A_i, B_j)) are removed from it (line 7).

The *overallSimilarity* between two classes is a normalized value (between 0 and 1) of their structural similarity by their syntactic similarity; it is computed using the following formula:

$$\frac{S_{Syntactic}+S_{Structural}}{1+S_{Syntactic}} \quad (1)$$

Where $S_{Syntactic}$ and $S_{Structural}$ represent the syntactic and structural similarities respectively.

Given two strings X and Y, the syntactic similarity $S_{Syntactic}$ between them is obtained by taking the *maximum* value from the *LCS* and *2-gram* algorithms respectively. This is motivated by three reasons. The first is that *LCS* does not provide accurate results in case of a change in word order. For example, *itemsPurchased* and *purchasedItems* are 0.643 similar based on *LCS* while *2-gram* provide better results (0.923 in this case). The second is that *2-gram* algorithm does not provide accurate results in case of short strings, e.g. *cr* and c*rd* are 0.66 similar based on *2-gram* while *LCS* provides better results (0.8 in this case). The third is that the *2-gram* algorithm does not provide accurate results in case of long strings where a substring has been replaced by a different word. For example, the similarity between *PurchasedItems* and *ShoppingCartItems* is 0.276 based on *2-gram* while *LCS* provides better a result (0.323 in this case). Consequently, taking the maximum value derived from both algorithms could give good early mappings provided the computation is case-insensitive and all the non-relevant characters such as space are not taken into account. For both algorithms, the similarity metric is defined as:

$$\frac{2*Length_{Same}}{Length_{All}} \quad (2)$$

For *LCS, $Length_{Same}$* is the cardinality of the set comprising all similar characters between X and Y. $Length_{Same}$ is the cardinality of the set comprising all similar substrings of size 2 obtained from X and Y respectively. $Length_{All}$ is the cardinality of the set comprising the disjoint union of the characters of both strings for *LCS*. Whereas, for *2-gram*, $Length_{All}$ is the cardinality of the set comprising the disjoint union of the substrings of size 2 obtained from both strings. For example, the syntactic similarity between the strings *ShoppingCart* and *SCart* is 0.471, which is the maximum value returned from LCS and 2-gram algorithms (0.471 and 0.4) respectively.

Given two classes or two class' elements such as operations and relationships, $S_{Structural}$ is calculated using the following formula:

$$\frac{2*sum}{sum+count} \quad (3)$$

Where *sum* is obtained by cumulating the syntactic similarities between all the *compatible* elements of the two classes of interest (or the items of the two elements of interest), and *count* is the number of elements (or items) used in the calculation of *sum*. The following is a list of basic elements/items taken into account when computing $S_{Structural}$.

- The class' name – applies to classes, attributes, operations, and all kind of relationships.
- The class' visibility list (public members) – applies to classes, attributes and operations.
- The class' ancestor(s) / descendent (s) – applies to classes.
- The class(es) aggregating (or composing) the class – applies to classes.
- The class' neighbor(s) (sibling(s)) – applies to classes and may also be applied to operations.
- The class' attributes – applies to classes.
- The type of an attribute – applies to class' attributes.
- The class' Invariant (predicate) – applies to classes.
- The class' Init (predicate) – applies to classes.
- The name of an operation – applies to operations.
- The list of Inputs / Outputs of an operation– applies to operations.
- The pre/post conditions (predicate) of an operation – applies to operations.

During the computation of $S_{Structural}$ for classes and operations, attribute names, inputs and outputs are replayed by their respective type. In addition, the same technique is applied to predicates (Inits, invariants, preconditions, and postconditions). The reason behind this is that for all the latter elements; type is the most important factor; names as well as their order of appearance could be ignored. Thus, the impact of $S_{Syntactic}$ on $S_{Structural}$ is reduced for behaviorally similar classes and operations. For example to compute the structural similarity between two operations, the names of the two classes containing the operations, the names of the two operations, the lists of the operations' inputs/outputs (replaced by their respective type), and their pre/post conditions (where names are replaced by their respective type) could be used. Even if the operations names, inputs and outputs are poorly syntactically similar, $S_{Structural}$ is capable of providing a more accurate value showing how much they are structurally (behaviorally) similar. $S_{Structural}$ for the operations *Add* and *New* (Figure 2) is 0.604 even if the syntactic similarity between their names is *null*.

The structural similarity of class' elements could be re-enforced once the class' overall similarity is computed. This concept is called mutual enforcing relationship and it is discussed in the following section along with other concepts intended to improve similarity detection.

IV. Improving Similarity Detection

A similarity detection approach that provides good balanced results (precision vs. recall) should integrate the

benefits of several methods because no particular method is better than others are. Thus, four additional concepts that could be used to strengthen the proposed similarity approach have been identified.

The first concept is the use of early landmarks. Assuming that the first view in figure 3 is created before the second view, after mining the specification repository, the developer of the second view could decide to create landmarks for classes/relationships that should be compared between the first view and the one he/she is about to create. These landmarks represent the classes/relationships that are most likely to match. This could greatly help improving the approach's efficiency. For example, if the developer of the second view decides to create landmarks between the classes {*ShoppingCart*, *CreditCard*} and {*SCart*, *CCard*} respectively, this reduces the number of class' comparisons from 20 down to 8 (i.e. 60% improvement in the time efficiency). In addition, the created landmarks improve the precision of the matching approach by reducing the number of false positives; i.e. less comparisons reduces the probability of additional false positives.

The second concept is automatically indexing the classes of the specifications created. An index could be the most frequent attribute(s) used in a class, and if used in the computation of structural similarity, it could improve the matching results. The other option is to use classes' descriptions (in natural language) that could be used when computing structural similarity. However, this option requires extra storage space as well as extra processing resources. Thus, automatic indexing seems to be a better option. For example, *items* could be used as an index for the *ShoppingCart* and *SCart* classes respectively. Thus, the similarity of their indexes is 1, which enhances their similarity scoring.

The third concept is the use *Mutual-Enforcing-Relationship* concept, i.e. after computing the overall similarity between classes, the results is used back to re-compute the similarity of the classes' elements. In other word, *classes are similar if they have similar elements and elements are similar if they are contained in similar classes*. This could improve the matching results of attributes, operations and relationships.

The final concept is the use of neighbors (siblings) similarity when computing structural similarity. For example when computing the structural similarity between the classes *CreditCard* and *CCard*, their neighbors (i.e. *PurchasedItem* and *SCard*) could be taken into account in the computation. The same concept could be applied to operations. However, the challenge is that the classes (and operations) order is not important, which makes choosing a suitable neighbor problematic. To solve this problem, the class (or operation) with the best possible match could be used as a sibling to all the classes (or operations) whose structural similarities have yet to be computed. For example, in figure 2, when computing the structural similarity between *Remove* and *Cancel*, the operations *Add* and *New* could be used as their neighbors respectively. The next section empirically evaluates the proposed approach and the impact of the introduced similarity improvement concepts on its performance.

V. EMPIRICAL EVALUATION

It has been argued that in case of small models developers may find it easy to identify the similarities manually. However, the proposed approach is intended to provide a quick and accurate way to identify matches when manual matching is not possible (or hard to achieve), which is the case of OO specifications intended for medium or large-scale software. Moreover, even for small models, experience has shown that identifying similarities manually is very difficult and error prone, especially for formal specifications.

A matching approach is useful if it produces accurate results with cheap processing means (i.e. time & space). The complexity of the proposed matching approach is *O(mn)* where m and n represent the number of classes in the two specifications respectively. This complexity could be reduced if the matched class of the second specification (or the matched classes of both specifications) is discarded from the next rounds of comparisons. However, leaving both matched classes was chosen, as it is possible to find better matches during the next rounds. In addition, normalizing (automatically or by a domain expert) the correspondence relation *R* is proposed before the start of the merging process. Finally, the use of landmarks reduces of the complexity from *O(mn)* to *O(k+(m-k)*(n-k))* where *k* is the number of landmarks created. The matching approach is effective if it does not produce too many incorrect matches (false positives) and does not produce too many missed matches (false negatives). *Precision* and *recall* metrics were used in the evaluation. *Precision* measures quality and is the ratio of correct matches found to the total number of matches found. *Recall* measures coverage and is the ratio of the correct matches found to the total number of all correct matches.

The proposed approach has been intensively evaluated based on several small/medium sized case studies. One of them includes the views of figure 3 and the following table summarizes their characteristics. Precision and recall were computed for a threshold ranging from 0.5 up to 0.9. The following figure shows the results obtained using *Match* algorithm without/with the similarity improvement concepts.

The results obtained (figure 5(a)) are in line with the theoretical assumption that elements with strong syntactic similarity are more likely to match. The elements with low syntactic similarity such as *ShoppingCart* and *SCart* (0.471) where confirmed as match through structural similarity (for the latter case it was 0.7). The overall similarity between *ShoppingCart* and *SCart* was 0.799. Low threshold (0.5 up to 0.6) has resulted in good recall (77%-92%) and acceptable/good precision (67%-83%). Medium threshold (0.65 up to 0.75) has resulted in low/acceptable recall (38%-69%) and good precision (90%-100%). Finally, high threshold (0.8 up to 0.9) has resulted in low recall (8%-31%) and perfect precision (100%).

TABLE I
CHARACTERISTICS OF THE STUDIED SPECIFICATIONS

	#Classes	#Attributes	#Operations	#Relationships	Total	#Matches
View 1	5	10	12	4	31	Classes - 4 Attributes - 5 Relationships - 2 Operations - 7 **Total = 18**
View 2	4	7	8	3	22	

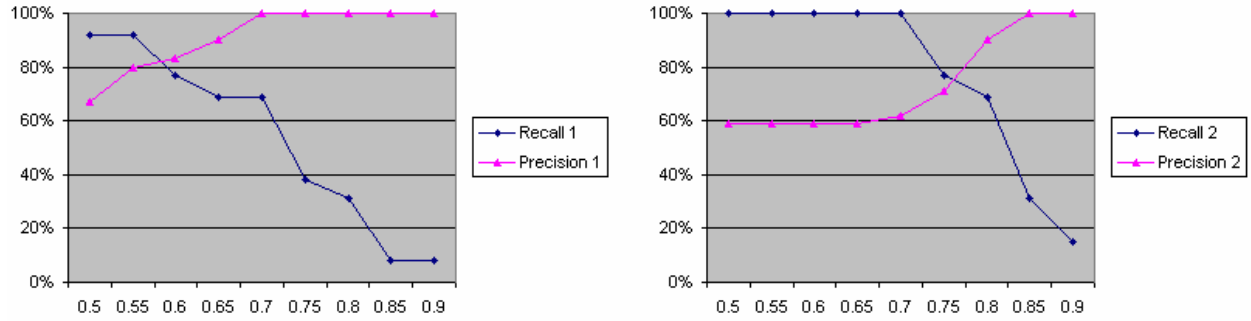

Fig. 5 Matching results for the two views of the online purchase system

Using the similarity improvement concepts of the previous section, the results (figure 5(b)) have showed a significant improvement in recall. Perfect recall (100%) was obtained for thresholds ranging from 0.5 up to 0.7. However, the improvements made on the method's recall have slightly affected its precision. Precision was acceptable (59%-62) for thresholds ranging from 0.5 up to 0.7. High threshold (0.75 up to 0.9) resulted in good/perfect precision (71%-100%).

To assess the overall improvement made, a reference threshold of 0.7 (that is not too low or too high) was chosen and the results obtained were compared. The recall was enhanced from 69% to 100%; however, the precision has decreased from 100% down to 62%. As the proposed framework is interactive, a domain expert could be responsible for normalizing the matching results obtained knowing that no matches have been missed (because of the additional concepts used) and that 38% of the results are not accurate. It is much easier for a domain expert to remove incorrect matches compared to identify missed ones. Furthermore, only the best match result for attributes and operations could be considered; in this case, the precision improves from 62% to 90%, thus, only 10% of false positive need to be removed from the match results. Consequently, a matching approach with good recall and (at least) an acceptable precision provides the best similarity detection means for the proposed framework.

VI. Related Work

In [20], class diagrams obtained by reverse engineering from a java software system are used to detect structural changes between the designs of subsequent versions of an OO software. The proposed algorithm reports the differences between the models in terms of additions, removals, changes, and renamings. The similarity detection used combines name and structural similarity metrics where only the 2-gram algorithm was used for the first and the semantics of OO design domain for the latter. [1] presents another differencing algorithm for OO programs. It identifies matching classes and methods of two given programs. The similarity is performed starting from the class (interface) level then down to the method level and finally at the node level.

In [12], two operators (match and merge) for hierarchical Statecharts models management were proposed. The match operator makes use of static and behavioral properties, and the use of sanity checks for the match results obtained is proposed before merging the models. Indeed, it is reasonable to make the matching process interactive where user seeds are used to confirm the most obvious relations as well as to rectify incorrect ones.

In [9], an algorithm is proposed for matching data schemas. The algorithm takes two directed labeled graphs as input and produces as output a mapping between the corresponding nodes of the graphs. The computation of similarity is iterative and follows the idea that models are similar when their adjacent elements are similar. Thus, the similarity of two elements propagates to their respective neighbors. The mapping results could be humanly adjusted, and the number of adjustments made was used to evaluate the algorithm's accuracy.

In [19], a design pattern detection methodology is proposed. It is based on similarity scoring between graph vertices. The graph similarity algorithm used takes as input

both the system and the pattern graphs and computes similarity scores between their vertices. The concept of mutually reinforcing relationship was used in the computation. The method was evaluated on three open source projects, and the results have shown good precision mainly because the pattern descriptions focused only on essential information.

In [6], an algorithm is proposed to discover mapping between schema elements using a broad set of techniques. It argued the necessity of a generic match component that should be studied independently. The proposed approach incorporates linguistic and structure matching. The approach was evaluated on XML schema and compared to other approaches from the literature. The approach has showed an improvement in mapping (similar) schema elements with low linguistic similarity.

Finally, [11] proposed a method for retrieving segments of source code from large repository. Conceptual graphs were used to model source code. Given a source code snippet and a repository of (source code) files, all documents of the repository are ranked according to their similarity to the code snippet. The similarity measure proposed exploits both structure and content, and is based on notion of contextual similarity. According to the latter, concepts should be compared not only by taking into account the information contained in them, but also by making use of the information contained in the concepts related to them (i.e. context).

The proposed approach is inspired from both [20] and [12]; however, the use of additional concepts that have shown to enhance similarity detection such as the use of landmarks, indexes, mutual-enforcing relationships, and neighbors' similarity was exploited. In addition, using information from the classes related to the class being compared is an integral part of the proposed approach; this has an identical impact as contextual similarity. The approach focused on requirements specification rather than design artifacts or source code because it allows the detection of conflicts at an early stage, which improves the quality of the developed software. The combination of both LCS and 2-gram algorithms to compute syntactic similarity provides a solid foundation, when combined with structural similarity; it produces an accurate overall similarity. Finally, because of the generic aspects of OO software artifacts used, the proposed approach could have a wide range of applications.

VII. Conclusion and Future Work

In this paper, a framework for collaborative development of OO formal specifications is proposed along with a similarity detection approach. The approach is intended to be used for merging specification views and checking their consistency. The approach combines syntactic, structural similarities to perform the matching, and a set of additional concept were proposed and their impact on improving the approach's performance was investigated. The empirical results obtained incorporated good precision/recall combined with an acceptable complexity. The precision could be further improved if only the best match for attributes and operations is considered; however, this concept need to be further studied and evaluated.

Since developers could specify classes differently, a challenging problem is to cater for subset-hood, i.e. how to match an operation *A* that has been implemented as two operations *B* and *C* in another similar class. Moreover, how to match a class *X* that has been implemented as two classes *Y* and *Z* in another view. Finally, assigning weights to classes' elements based on their importance could play a vital role in similarity detection, but the basis on which weights are assigned needs to be investigated.

References

[1] T. Apiwattanapong et al, "A Differencing Algorithm for Object-Oriented Programs", in 2004 *Proc. 19th International Conference on Automated Software Engineering,* pp. 2-13.

[2] M. Godfrey et al., "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", *IEEE Transactions on Software Engineering*, Vol. 31, No. 2, pp. 166-181, 2005.

[3] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1999.

[4] G. Jeh et al., "SimRank: A Measure of Structure-Context Similarity", in 2002 *Proc. International Conference on Knowledge Discovery and Data Mining*, pp. 538-543.

[5] G. Kassel et al., "Model Checking Object-Z classes: Some Experiments with FDR", in *Proc. APSEC conference*, pp. 445-452.

[6] J. Madhavan et al., "Generic Schema Matching with Cupid", in 2001 *Proc. 27th VLDB conference.*

[7] C. Manning et al., *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.

[8] A. Mehra et al., "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design", in 2005 *Proc. International Conference on Automated Software Engineering*, pp. 204-213.

[9] S. Melnik et al., "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching", in 2002 *Proc. International Conference on Data Engineering*, pp. 117-128.

[10] T. Mens, "A State-of-the-Art Survey on Software Merging", *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, pp. 449-462, 2002.

[11] G. Mishne et al., "Source Code Retrieval using Conceptual Similarity", in 2004 *Proc. RIAO 2004 Conference*, pp. 539-554.

[12] S. Nejati et al., "Matching and Merging of Statecharts Specifications", in 2007 *Proc. 29th International Conference on Software Engineering (ICSE'07)*.

[13] A. Boronat et al., "Formal Model Merging Applied to Class Diagram Integration", *Electronic Notes in Theoretical Computer Science*, Vol. 166, pp. 5–26, 2007.

[14] B. Nuseibeh et al., "Making Consistency Respectable in Software Development", *Journal of Systems and Software*, Vol. 58, pp. 171-180, 2001.

[15] L. Prechelt et al., "JPlag: Finding Plagiarisms Among a Set of Programs", Department of Informatics, University of Karlsruhe, Tech. Rep. No. 1, March 2000.

[16] M. Sabetzadeh et al., "Consistency Checking of Conceptual Models via Model Merging", in 2007 *Proc. 15th IEEE RE conference.*

[17] G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.

[18] J. Spivey, *The Z notation – A Reference Manual*, Prentice Hall, 2nd Edition, 1992

[19] N. Tsantalis et al., "Design Pattern Detection Using Similarity Scoring", *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, pp. 896-909, 2006.

[20] Z. Xing et al., "UMLDiff: An Algorithm for Object-Oriented Design Differencing", in 2005 *Proc. 20th IEEE/ACM international Conference on Automated software engineering*, pp. 54-65.